

Разработка сложного программного обеспечения (ПО) в кратчайшие сроки и с наименьшими затратами требует правильной организации всего процесса создания ПО. Инженерно-техническую дисциплину, связанную с *созданием ПО* и *поддержанием его в рабочем состоянии* называют "программная инженерия". В нее входят анализ и постановка задачи, построение алгоритмов и разработка структур данных, проектирование программ и написание кода, отладка и тестирование ПО, документирование, сопровождение и многие другие организационные и технические стороны создания ПО. Все эти процессы исследуются и постоянно совершенствуются в рамках дисциплин *конструирование ПО* и *Технология программирования*.

*Конструирование ПО* – это детальное создание работающего программного обеспечения посредством комбинации кодирования, верификации, модульного тестирования, интеграционного тестирования и отладки.

*Технология программирования* – совокупность методов и средств, используемых в процессе разработки программного обеспечения. Как и любая другая, технология программирования представляет собой набор правил и инструкций, включающих:

- перечень и указание последовательности выполнения технологических операций;
- перечисление условий, при которых выполняется та или иная операция;
- описания самих операций, где для каждой операции определены исходные данные, результаты, а также инструкции, нормативы, стандарты, критерии и методы оценки и т. п.

Сегодня разработка ПО – серьезный и ресурсоемкий бизнес. Существенные доходы могут быть получены только при условии предоставления пользователям таких новых возможностей, за которые они готовы будут заплатить деньги. Для этого при составлении технического задания на разработку ПО необходимо учитывать многое: например, производительность процессоров и объем оперативной памяти в компьютерах потенциальных пользователей, скорость и объем доступного интернет трафика, удобство интерфейса, экономическую целесообразность применения ПО или высокое качество развлечений и т.п.

Большая часть трудностей при разработке ПО связана, в том числе, с организацией совместной работы многих людей, возможно удаленных друг от друга географически. Причем работа этих людей должна быть организована так, чтобы затраты на разработку были меньше доходов от продажи ПО. Рассчитывая стоимость разработки ПО необходимо учитывать, что в затраты входит не только зарплата разработчиков, затраты на оборудование и инструменты разработки, приобретение лицензий. Необходимо тратить деньги и на исследование потребностей клиентов и на проведение рекламы и другой маркетинговой деятельности.

Необходимо учитывать также, что практически значимые программные системы всегда содержат ошибки, т.е. нельзя утверждать, что она всегда правильно решает все поставленные перед ней задачи. Этот факт связан с практической невозможностью доказательства отсутствия ошибок или полной проверки работы ПО. Поэтому разработка сложного ПО связана с нахождением разумного компромисса между затратами на разработку и качеством ее результата. В затраты входят все виды используемых ресурсов, из которых наиболее важны затраченное время и бюджет проекта. Удовлетворение пользователей от работы с программой определяется термином *качество программы*, которое включает в себя набор предоставляемых возможностей, надежность, удобство использования, гибкость, удобство внесения изменений и исправления ошибок.

Создание качественного ПО в заданные сроки и с минимальной стоимостью возможно лишь при использовании современной *технологии программирования*.

# ЛАБОРАТОРНАЯ РАБОТА № 1

## Этапы разработки программного обеспечения. Договор на разработку. Техническое задание

**Цель работы:** ознакомиться с процессом заключения договора на разработку программного обеспечения (ПО) и правилами подготовки технического задания (ТЗ) на разработку ПО.

**Продолжительность работы – 4 часа.**

### Содержание

Теория. Процесс заключения договора.....	2
Порядок разработки ТЗ .....	4
Порядок выполнения лабораторной работы.....	8
Вопросы.....	8
Приложения .....	9

### 1. Процесс заключения договора

Создание программного обеспечения – процесс дорогой и ресурсоемкий. Любой продукт (не только ПО!) создается либо по чьему-то заказу, либо за счет внутренних ресурсов компании с перспективой дальнейшей продажи. Чаще всего создание ПО выполняется компаниями-разработчиками по заказам других организаций и начинается этот процесс с заключения договора на создание продукта.

Необходимо понимать, что заключение договора – процесс юридический, подробно определен законодательством РФ (ст. 432 ГК) и предполагает две стадии:

1. предложение (оферта);
2. ее принятие (акцепт).

Соответственно можно выделить следующие стадии заключения договора:

- преддоговорные контакты сторон (переговоры);
- оферта;
- рассмотрение оферты;
- акцепт оферты.

При этом две стадии — оферта и акцепт оферты — являются обязательными для всех случаев заключения договора.

Стадия *преддоговорных контактов* сторон (переговоров) носит факультативный характер и используется по усмотрению сторон, вступающих в договорные отношения. На этапе преддоговорного периода стороны формулируют стоящие перед ними экономические задачи, которые предстоит реализовать.

*Оферт* (оформленной документально, направленной и полученной адресатом) присуще еще одно важное свойство — безотзывность. Принцип безотзывности оферты, т.е. невозможности для оферента отзывать свое предложение о заключении договора в период с момента получения его адресатом и до истечения установленного срока для ее акцепта, сформулирован в виде презумпции (ст. 436 ГК).

Для того чтобы договор был признан заключенным, необходим полный и безоговорочный *акцепт*, т.е. согласие лица, получившего оферту, на заключение договора на предложенных в оферте условиях (ст. 443 ГК).

Момент заключения договора определяет время его вступления в силу, т.е. обязательность для сторон условий заключенного договора (п. 1 и 2 ст. 425 ГК). Именно с этого момента исполнитель несет полную финансовую и правовую ответственность за исполнение принятых на себя обязательств.

Разработка ПО – это всегда новое и штучное. Даже выпуск новой версии уже существующего ПО – это всегда развитие и совершенствование. Разработка ПО – это всегда дорого и сложно. Поэтому при заключении договора на разработку необходимо учитывать не только желание заказчиков, но и технические (опыт и навыки коллектива разработчиков, материальное обеспечение) и организационные возможности компании.

Неотъемлемой частью договора являются следующие документы (оформляемые как приложения к договору):

- техническое задание;
- план-график выполнения работ;
- классификация дефектов и сроки их устранения.

По общепринятым на сегодня как международным так и Российской стандартам процесс разработки включает следующие действия:

- *анализ требований к системе* – определение ее функциональных возможностей, пользовательских требований, требований к надежности и безопасности, требований к внешним интерфейсам и т. д.;
- *подготовительную работу* – выбор модели жизненного цикла (см. далее), стандартов, методов и средств разработки, а также составление плана работ;
- *проектирование структуры системы* – определение состава необходимого оборудования, программного обеспечения и операций, выполняемых обслуживающим персоналом;
- *анализ требований к программному обеспечению* – определение функциональных возможностей, включая характеристики производительности, среды функционирования компонентов, внешних интерфейсов, спецификаций надежности и безопасности, эргономических требований, требований к используемым данным, установке, приемке, пользовательской документации, эксплуатации и сопровождению;
- *детальное проектирование программного обеспечения* – подробное описание компонентов программного обеспечения и интерфейсов между ними, обновление пользовательской документации, разработка и документирование требований к тестам и плана тестирования компонентов программного обеспечения, обновление плана интеграции компонентов;
- *кодирование и тестирование программного обеспечения* – разработку и документирование каждого компонента, а также совокупности тестовых процедур и данных для их тестирования, тестирование компонентов, обновление пользовательской документации, обновление плана интеграции программного обеспечения;
- *интеграцию программного обеспечения* – сборку программных компонентов в соответствии с планом интеграции и тестирование программного обеспечения на соответствие квалификационным требованиям, представляющим собой набор критериев или условий, которые необходимо выполнить, чтобы квалифицировать программный продукт, как соответствующий своим спецификациям и готовый к использованию в заданных условиях эксплуатации;
- *квалификационное тестирование программного обеспечения* – тестирование программного обеспечения в присутствии заказчика для демонстрации его соответствия требованиям и готовности к эксплуатации; при этом проверяется также готовность и полнота технической и пользовательской документации;

- *интеграцию системы* – сборку всех компонентов системы, включая программное обеспечение и оборудование;
- *квалификационное тестирование системы* – тестирование системы на соответствие требованиям к ней и проверка оформления и полноты документации;
- *установку программного обеспечения* – установку программного обеспечения на оборудовании заказчика и проверку его работоспособности;
- *приемку программного обеспечения* – оценку результатов квалификационного тестирования программного обеспечения и системы в целом и документирование результатов оценки совместно с заказчиком, окончательную передачу программного обеспечения заказчику.

Стадия *определения требований* к ПО – одна из важнейших и определяет в значительной степени успех всего проекта. Она включает следующие этапы:

- Планирование работ, предваряющее работы над проектом. Определение целей разработки, предварительная экономическая оценка проекта, построение плана-графика выполнения работ, создание и обучение рабочей группы. Иногда вводят дополнительно начальную стадию – анализ осуществимости системы.
- Проведение обследования объекта разработки, в рамках которого осуществляются предварительное выявление требований к будущей системе, определение структуры, определение перечня целевых функций, анализ распределения функций по подразделениям и сотрудникам, выявление функциональных взаимодействий между подразделениями, информационных потоков внутри подразделений и между ними, внешних по отношению к организации объектов и внешних информационных воздействий, анализ существующих средств автоматизации деятельности.

*Техническое задание* представляет собой документ, в котором сформулированы основные цели разработки, требования к программному продукту, определены сроки и этапы разработки и регламентирован процесс приемо-сдаточных испытаний. В разработке технического задания участвуют как представители заказчика, так и представители исполнителя. В основе этого документа лежат исходные требования заказчика, анализ передовых достижений техники, результаты выполнения научно-исследовательских работ, предпроектных исследований, научного прогнозирования и т.п. В зависимости от сложности разрабатываемого ПО, *разработка технического задания* может быть как этапом разработки, так и четкими условиями заключенного договора.

## 2. Порядок разработки технического задания

Разработка технического задания выполняется в следующей последовательности.

Прежде всего, проводят анализ требований к ПО. *Требование* — это любое условие, которому должна соответствовать разрабатываемая система или программное средство. *Требованием* может быть возможность, которой система должна обладать и ограничение, которому система должна удовлетворять.

В соответствии с Глоссарием терминов программной инженерии IEEE ITILv3 (IT Infrastructure Library) [43], являющимся общепринятым международным стандартным глоссарием, *требование* это:

1. Условия или возможности, необходимые пользователю для решения проблем или достижения целей;
2. Условия или возможности, которыми должна обладать система или системные компоненты, чтобы выполнить контракт или удовлетворять стандартам, спецификациям или другим формальным документам;

3. Документированное представление условий или возможностей для пунктов 1 и 2.

В соответствии со стандартом разработки *требований* ISO/IEC 29148, *требование* – это утверждение, которое идентифицирует эксплуатационные, функциональные параметры, характеристики или ограничения проектирования продукта или процесса, которое однозначно, проверямо и измеримо. *Требования* необходимы для приемки продукта или процесса (потребителем или внутренним руководящим принципом обеспечения качества).

Глоссарий ITILv3 определяет такое понятие, как *набор требований* – документ, содержащий все *требования* к продукту, а также к новой или измененной ИТ-услуге.

*Требование* должно обладать следующими характеристиками:

1. *Единичность* — требование описывает одну и только одну вещь.
2. *Завершенность* — требование полностью определено в одном месте и вся необходимая информация присутствует.
3. *Последовательность* — требование не противоречит другим требованиям и полностью соответствует документации.
4. *Атомарность* — требование нельзя разделить на более мелкие.
5. *Отслеживаемость* — требование полностью или частично соответствует деловым нуждам как заявлено заинтересованными лицами и задокументировано.
6. *Актуальность* — требование не стало устаревшим с течением времени.
7. *Выполнимость* — требование может быть реализовано в рамках проекта.
8. *Недвусмысленность* — требование определено без обращения к техническому жаргону, акронимам и другим скрытым формулировкам. Оно выражает объекты и факты, а не субъективные мнения. Возможна одна и только одна его интерпретация. Определение не содержит нечетких фраз, использование отрицательных и составных утверждений запрещено.
9. *Обязательность* — требование представляет собой определенную заинтересованым лицом характеристику, отсутствие которой ведет к неполноценности решения, которая не может быть проигнорирована. Необязательное требование — противоречие самому понятию требования.
10. *Проверяемость* — реализованность требования может быть проверена.

В соответствии с ITILv3 все требования в проекте можно разделить на следующие группы:

1. *Функциональные* — реализуют саму бизнес-функцию.
2. *Управленческие* — требования к доступным и безопасным сервисам; относятся к размещению системы, администрированию и безопасности.
3. *Эргономические* — к удобству работы конечных пользователей.
4. *Архитектурные* — требования к архитектуре системы.
5. *Взаимодействия* — к взаимосвязям между существующими приложениями и программными средствами и новым приложением.
6. *Сервисного уровня* — описывают поведение сервиса, качество его выходных данных и другие качественные аспекты, измеряемые заказчиком.

Важно:

Спецификация требований не содержит деталей дизайна или реализации (кроме известных ограничений), данных о планировании проекта или сведений о тестировании. Это относится к требованиям к проекту, а не к продукту и формируются на следующих стадиях разработки ПО.

Далее устанавливают набор выполняемых функций, а также перечень и характеристики исходных данных. Затем определяют перечень результатов, их характеристики и способы представления.

Далее уточняют среду функционирования программного обеспечения: конкретную комплектацию и параметры технических средств, версию используемой операционной системы и, возможно, версии и параметры другого установленного программного обеспечения, с которым предстоит взаимодействовать будущему программному продукту.

В случаях, когда разрабатываемое программное обеспечение собирает и хранит некоторую информацию или включается в управление каким-либо техническим процессом, необходимо также четко регламентировать действия программы в случае сбоев оборудования и энергоснабжения.

### ***Общие положения по оформлению ТЗ***

Техническое задание в рамках лабораторной работы оформляют в соответствии с ГОСТ 19.106-78 на листах формата А4.

Титульный лист оформляют в соответствии с ГОСТ 19.104-78. 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.

Техническое задание должно содержать следующие разделы:

- введение;
- наименование 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 приёмки;
- приложения.

В зависимости от особенностей программы или программного изделия допускается уточнять содержание разделов, вводить новые разделы или объединять отдельные из них. При необходимости допускается в техническое задание включать приложения.

### ***Содержание разделов***

Введение должно включать краткую характеристику области применения программы или программного продукта, а также объекта (например, системы) в котором предполагается их использовать. Основное назначение введения – продемонстрировать актуальность данной разработки и показать, какое место эта разработка занимает в ряду подобных.

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.

В разделе "Основание для разработки" должны быть указаны:

- документ (документы), на основании которых ведется разработка. Таким документом может служить план, приказ, договор и т. п.;
- организация, утвердившая этот документ, и дата его утверждения;

- наименование и (или) условное обозначение темы разработки.

В разделе "Назначение разработки" должно быть указано функциональное и эксплуатационное назначение программы или программного изделия.

Раздел "Технические требования к программе или программному изделию" должен содержать следующие подразделы:

- требования к функциональным характеристикам;
- требования к надёжности;
- 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.

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

В подразделе "Требования к надёжности" должны быть указаны требования к обеспечению надёжного функционирования (обеспечение устойчивого функционирования, контроль входной и выходной информации, время восстановления после отказа и т.п.)

В подразделе "Условия эксплуатации" должны быть указаны условия эксплуатации (температура окружающего воздуха, 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.

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

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.

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

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.

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.

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 так же, как правило, сроки разработки и определяют исполнителей.

В разделе "Порядок контроля и приёмки" должны быть указаны виды испытаний и общие требования к приёмке работы.

В приложениях к техническому заданию, при необходимости, приводят:

- перечень научно-исследовательских и других работ, обосновывающих разработку;

- схемы алгоритмов, таблицы, описания, обоснования, расчёты и другие документы, которые могут быть использованы при разработке;
- другие источники разработки.

В случаях, если какие-либо требования, предусмотренные техническим заданием, заказчик не предъявляет, следует в соответствующем месте указать «Требования не предъявляются».

### **3. Порядок выполнения работы.**

1. Изучить теоретические сведения, приведенные в описании лабораторной работы.
2. Разработать техническое задание на программный продукт (см. варианты заданий в приложении 1).
3. Оформить работу в соответствии с ГОСТ 19.106-78. При оформлении использовать MS Office. Примеры оформления ТЗ приведены в приложении 2 и 3.
4. Сдать и защитить работу.

Отчет по лабораторной работе должен включать в себя:

1. Постановку задачи.
2. Техническое задание на программный продукт.

Защита отчета по лабораторной работе заключается в предъявлении преподавателю полученных результатов (на экране монитора), демонстрации полученных навыков и ответах на вопросы преподавателя.

### **Контрольные вопросы**

1. Какие процедуры выполняются при заключении договора.
2. Этапы разработки программного обеспечения.
3. Постановка задачи и предпроектные исследования.
4. Функциональные и эксплуатационные требования к программному продукту.
5. Правила разработки технического задания.
6. Основные разделы технического задания.

## ПРИЛОЖЕНИЕ 1.

### ВАРИАНТЫ ЗАДАНИЙ НА ЛАБОРАТОРНУЮ РАБОТУ

Для отработки навыков коллективной разработки ПО лабораторной работы выполняются группами студентов от 2 до 5 человек. Лабораторные работы №№ 1-8 выполняются каждой группой для одного и того же варианта.

Примеры, приведенные в перечне ниже, не являются обязательными. Фантазия студентов в выборе темы приветствуется и ничем, кроме возможности реализации, не ограничивается.

1. Разработать программный модуль «Система контроля ж/д переезда». Программный модуль предназначен для управления движением автомобилей через железнодорожный переход.
2. Разработать программный модуль «Касса в магазине». Программный модуль предназначен для контроля штрих-кодов товаров, учета расходования товара и коррекции склада.
3. Разработать программный модуль «Система контроля взлета-посадки». Модуль должен отслеживать текущую обстановку на территории аэродрома и учитывать воздушную обстановку.
4. Разработать программный модуль «Электронный секретарь». Программа предназначена для записи мероприятий, предупреждения в реальном времени о плановых мероприятиях, отслеживания юбилеев коллег и партнеров.
5. Разработать приложение для поликлиник «Электронная запись к врачу». Приложение предназначено для больных, желающих попасть на прием как в день обращения, так и на месяц вперед.
6. Разработать программный модуль «Система заказа фотографий», содержащий возможности заказать фотографии любого размера на разных видах фотобумаги с различных носителей информации.
7. Разработать программный модуль «Умный дом». Модуль предназначен для удаленного управления загородным домом.
8. Разработать программный модуль «Бронирование авиабилетов в турфирмах». Необходимо обеспечить возможность бронирования билетов как прямых рейсов, так и перелетов с учетом стыковки рейсов разных авиакомпаний.
9. Разработать программный модуль «Автоматизированная система поздравления друзей и коллег». Должен учитываться пол, возраст и увлечения. Поздравления не должны быть одинаковыми и не могут повторяться для людей знакомых друг с другом.

*Примечание: При разработке программы не ограничивайтесь функциями, приведенными в варианте. Подойдите к задаче творчески – постарайтесь предусмотреть расширение возможностей проектируемой программы. Обязательно использование структурного и модульного подхода к программированию. Желательно использование объектного подхода.*

**ПРИЛОЖЕНИЕ 2.**

**ПРИМЕР РАЗРАБОТКИ ТЕХНИЧЕСКОГО ЗАДАНИЯ  
НА ПРОГРАММНЫЙ ПРОДУКТ**

Министерство образования Российской Федерации

Московский государственный институт электронной техники

(технический университет)

Кафедра Информатики и программного обеспечения вычислительных  
систем

УТВЕРЖДАЮ

Зав. Кафедрой ИПОВС,

д.т.н., проф. \_\_\_\_\_ Гагарина Л.Г.

«\_\_»\_\_\_\_\_ 2015 г.

ПРОГРАММА СОРТИРОВКИ ОДНОМЕРНОГО МАССИВА

Техническое задание на лабораторную работу

Листов 3

Руководитель, к.т.н., доцент \_\_\_\_\_ Петров А.А.

Исполнитель, студент гр. МП 33 \_\_\_\_\_ Власов С.Е.

МОСКВА, 2015

*Рисунок 2.1. Пример оформления титульного листа технического задания на учебный  
программный продукт*

## **1. Введение**

Настоящее техническое задание распространяется на разработку программы сортировки одномерного массива методами пузырька, прямого выбора, Шелла и быстрой сортировки, предназначеннной для использования школьниками старших классов при изучении курса школьной информатики.

## **2. Основание для разработки**

- 2.1. Программа разрабатывается на основе учебного плана кафедры «Информатики и программного обеспечения вычислительных систем»
- 2.2. Наименование работы  
«Программа сортировки одномерного массива»
- 2.3. Исполнитель: компания BestSoft.
- 2.4. Соисполнители: нет.

## **3. Назначение**

Программа предназначена для использования школьниками при изучении темы «Обработка одномерных массивов» в курсе «Информатика».

## **4. Требования к программе или программному изделию**

- 4.1. Требования к функциональным характеристикам
  - 4.1.1. Программа должна обеспечивать возможность выполнения следующих функций:
    - ввод размера массива и самого массива;
    - хранение массива в памяти;
    - выбор метода сортировки;
    - вывод текстового описания метода сортировки;
    - вывод результата сортировки.
  - 4.1.2. Исходные данные:
    - размер массива, заданный целым числом;
    - массив;
  - 4.1.3. Организация входных и выходных данных  
Входные данные поступают с клавиатуры.  
Выходные данные отображаются на экране и при необходимости выводятся на печать.
- 4.2. Требования к надежности  
Предусмотреть контроль вводимой информации.  
Предусмотреть блокировку некорректных действий пользователя при работе с системой.
- 4.3. Требования к составу и параметрам технических средств  
Система должна работать на IBM совместимых персональных компьютерах.  
Минимальная конфигурация:
  - тип процессора – Pentium и выше;
  - объем оперативного запоминающего устройства – 32 Мб и более;
  - объем свободного места на жестком диске – 40 Мб.  
Рекомендуемая конфигурация:
  - тип процессора – Intel Core 5;
  - объем оперативного запоминающего устройства – 512 Мб;
  - объем свободного места на жестком диске – 600 Мб.

#### **4.4. Требования к программной совместимости**

Программа должна работать под управлением семейства операционных систем Win 32 (Windows 95/98/2000/ME/XP и т. п.).

### **5. Требования к программной документации**

- 5.1. Разрабатываемые программные модули должны быть самодокументированы, т. е. тексты программ должны содержать все необходимые комментарии.
- 5.2. Разрабатываемая программа должна включать справочную информацию о работе программы, описания методов сортировки и подсказки учащимся.
- 5.3. В состав сопровождающей документации должны входить:
  - 5.3.1. Пояснительная записка на 5 листах, содержащая описание разработки.
  - 5.3.2. Руководство пользователя.

**ПРИЛОЖЕНИЕ 3.**

**ПРИМЕР ТЕХНИЧЕСКОГО ЗАДАНИЯ НА РАЗРАБОТКУ**

«Утверждаю»

Профессор кафедры ВС

\_\_\_\_\_ (Иванов И.И.)

«\_\_\_» \_\_\_\_\_ 2015 г.

**Техническое задание**  
на разработку «Модуля автоматизированной  
системы оперативно-диспетчерского управления  
теплоснабжением корпусов МИЭТ»

**Москва, 2015**

## **1. Введение**

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

## **2. Основание для разработки**

- 2.1. Основанием для данной работы служит договор № 1234 от 10 марта 2014 г.
- 2.2. Наименование работы «Модуль автоматизированной системы оперативно-диспетчерского управления теплоснабжением корпусов МИЭТ».
- 2.3. Исполнители: ОАО “Лаборатория создания программного обеспечения”
- 2.4. Соисполнители: нет.

## **3. Назначение разработки**

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

## **4. Технические требования**

### **4.1. Требования к функциональным характеристикам**

#### **4.1.1. 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 настройки;

В конце отчетного периода система должна архивировать данные.

#### **4.1.2. Организация входных и выходных данных**

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

вентиляцию в помещениях. Возможно также автоматическая установка некоторых параметров для устройств регулирования.

Основной режим использования системы – ежедневная работа.

#### 4.2. Требования к надежности

Для обеспечения надежности необходимо проверять корректность получаемых данных с датчиков.

#### 4.3. 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.

#### 4.4. Требования к информационной и программной совместимости

Программа должна работать на платформах Windows 98/NT/2000

#### 4.5. Требования к транспортировке и хранению

Программа поставляется на лазерном носителе информации. Программная документация поставляется в электронном и печатном виде.

#### 4.6. Специальные требования

- программное обеспечение должно иметь дружественный интерфейс, рассчитанный на пользователя (в плане компьютерной грамотности) квалификации;
- ввиду объемности проекта, задачи предполагается решать поэтапно, при этом модули ПО, созданные в разное 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 (например счетчик SA-94 и т.п.)

### **5. Требования к программной документации**

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

### **6. Технико-экономические показатели**

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

### **7. Порядок контроля и приемки**

После передачи Исполнителем отдельного функционального модуля программы Заказчику, последний имеет право тестировать модуль в течении 7 дней. После

тестирования Заказчик должен принять работу по данному этапу или в письменном виде изложить причину отказа принятия. В случае обоснованного отказа Исполнитель обязуется доработать модуль.

## 8. Календарный план работ

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

Руководитель работ

Сидоров С.В.

## **ПРИЛОЖЕНИЕ 4.**

### **ПРИМЕР ТИПОВОГО ДОГОВОРА НА РАЗРАБОТКУ ПО**

ДОГОВОР № \_\_\_\_\_  
на разработку программного обеспечения

г. Москва

" \_\_\_\_ " 20 \_\_\_\_ г.

\_\_\_\_\_ (полное и сокращённое наименование заказчика), именуемый в дальнейшем Заказчик, в лице \_\_\_\_\_ (должность, фамилию, имя, отчество представителя) \_\_\_\_\_, действующего на основании \_\_\_\_\_ (наименование и реквизиты документа, на основании которого действует представитель) \_\_\_\_\_, с одной стороны, и \_\_\_\_\_ (полное и сокращённое наименование контрагента) \_\_\_\_\_, именуем \_\_\_\_\_ в дальнейшем Исполнитель, в лице \_\_\_\_\_ (указать должность, фамилию, имя, отчество представителя) \_\_\_\_\_, действующего на основании \_\_\_\_\_ (указать наименование и реквизиты документа, на основании которого действует представитель) \_\_\_\_\_, с другой стороны, совместно именуемые далее Стороны, а каждая в отдельности Сторона, заключили настоящий договор (далее – Договор) о нижеследующем.

#### **1. ПРЕДМЕТ ДОГОВОРА**

1.1. Исполнитель обязуется выполнить работы по разработке \_\_\_\_\_[1] программного обеспечения \_\_\_\_\_[2] (далее ПО), в том числе его компонент ( \_\_\_\_\_)[3], а также оказать услуги по гарантийной технической поддержке разработанного по настоящему Договору ПО. С момента передачи Исполнителем Заказчику результатов работ по разработке ПО в порядке, указанном в п.3.1. Договора, исключительное право на ПО в полном объеме без ограничений принадлежит Заказчику. Требования к ПО, а также представляемые Исполнителем результаты работ по разработке ПО, определяются в Техническом задании (Приложение ) (далее – ТЗ), которое подписывается Сторонами и является неотъемлемой частью Договора. Результаты работ Исполнитель передает на материальном носителе ( \_\_\_\_\_)[4].

1.2. Перечень и сроки выполнения работ определяются в Плане-графике выполнения работ (Приложение ), который подписывается Сторонами и является неотъемлемой частью Договора.

1.3. Заказчик обязуется принять и оплатить работы, выполненные Исполнителем по Договору.

#### **2. СТОИМОСТЬ И ПОРЯДОК РАСЧЕТОВ**

2.1. Стоимость работ по разработке ПО составляет \_\_\_\_\_ (\_\_\_\_) \_\_\_, кроме того НДС (18%) – \_\_\_\_\_ (\_\_\_\_) \_\_\_, итого с учетом НДС – \_\_\_\_\_ (\_\_\_\_) \_\_\_. В стоимость работ по Договору включена стоимость материальных носителей на которых передаются результаты работ.

2.2. Заказчик производит оплату работ по разработке ПО в соответствии с Графиком платежей (Приложение к Договору) на основании счетов, представленных Исполнителем.

2.3. Оплата работ производится в российских рублях путем перечисления Заказчиком денежных средств на расчетный счет Исполнителя, указанный в Статье 15 Договора. Датой исполнения обязательств Заказчика по платежам считается дата списания денежных средств со счета Заказчика.

2.4. В платежно-расчетных документах НДС выделяется отдельной строкой.

2.5. Исполнитель предоставляет Заказчику счета-фактуры в порядке и сроки, установленные законодательством Российской Федерации.

#### **3. ПОРЯДОК СДАЧИ-ПРИЕМКИ РАБОТ**

3.1. По завершении выполнения работ Исполнитель уведомляет об этом Заказчика и передает Заказчику по Акту сдачи-приемки работ (в \_\_\_\_\_ (\_\_\_\_)[5] экземплярах) (форма Акта сдачи-приемки работ приведена в Приложении № 4 к Договору) полный комплект программ с технической документацией, дистрибутивов, других материалов и документов, предусмотренных ТЗ, на материальных носителях и счет на оплату. Заказчик обязан провести проверку, принять выполненные работы и подписать все экземпляры Акта сдачи-приемки работ, \_\_\_\_\_ (\_\_\_\_)[6] из которых направить Исполнителю в течение \_\_\_\_\_ (\_\_\_\_)[7] рабочих дней с даты получения, либо в этот же срок направить Исполнителю мотивированный отказ от подписания Акта.

3.2. При наличии мотивированного отказа Заказчика от подписания Акта Стороны в течение \_\_\_\_\_ (\_\_\_\_)[8] рабочих дней с даты получения Исполнителем мотивированного отказа согласовывают Протокол несоответствий по форме, предусмотренной Приложением № 6 к Договору, в котором

указываются также способы и сроки устранения замечаний. Выявленные несоответствия по согласованному Протоколу несоответствий устраняются Исполнителем в срок, предусмотренный в Протоколе несоответствий, без увеличения объема работ и без увеличения общей стоимости работ, указанной в п.2.1. Договора. После принятия Исполнителем мер для устранения замечаний процедура подписания Акта сдачи-приемки работ повторяется.

3.3. Исполнитель вправе по согласованию с Заказчиком досрочно выполнить работы.

#### **4. ВНЕСЕНИЕ ИЗМЕНЕНИЙ**

4.1. Любая из Сторон может потребовать внесения изменений в Договор, в том числе по условиям, срокам, составу и стоимости работ.

4.2. Требование Исполнителя о внесении изменений не является обязательным для Заказчика, если Исполнитель не докажет, что данное требование связано с технической невозможностью выполнить работы без изменения ТЗ.

4.3. Требование Заказчика о внесении изменений является обязательным для Исполнителя. Исполнитель в течение \_\_\_\_\_(\_\_\_\_)[9] рабочих дней со дня получения соответствующего требования направляет Заказчику смету по внесению соответствующих изменений с указанием стоимости работ и сроков их выполнения. Исполнитель может отказаться от исполнения требования Заказчика о внесении изменений только в случае технической невозможности их реализации.

4.4. Внесение изменений оформляется подписанием Сторонами Дополнительного соглашения (форма Дополнительного соглашения – Приложение к Договору).

#### **5. ОТВЕТСТВЕННОСТЬ СТОРОН**

5.1. Исполнитель обязуется выполнять работы, указанные в п.1.1. Договора, самостоятельно. При необходимости, по согласованию с Заказчиком, Исполнитель вправе привлекать к исполнению Договора третьи лица, без дополнительной оплаты Заказчиком, неся за них полную ответственность, в т. ч. по конфиденциальности предоставляемой информации, за качество выполнения работ, а также за убытки в порядке, предусмотренном в п. 5.13. Договора. Допуск представителей третьих лиц осуществляется в порядке, предусмотренном в п.5.12 Договора.

5.2. За неисполнение или ненадлежащее исполнение обязательств по Договору Стороны несут ответственность в соответствии с законодательством Российской Федерации.

5.3. В каждом случае нарушения любого из: сроков выполнения работ, установленных в Приложении № 2 к Договору, сроков устранения выявленных несоответствий, установленных в Протоколе несоответствий, сроков внесения изменений в соответствии с п.4.3 Договора, Исполнитель выплачивает Заказчику неустойку в размере \_\_\_\_\_(\_\_\_\_)[10] %, включая НДС, от общей стоимости работ, указанной в п.2.1. Договора, за каждый календарный день просрочки, но не более \_\_\_\_\_(\_\_\_\_)[11] % от этой суммы за каждый случай.

5.4. В случае нарушения сроков оплаты, установленных в Приложении № 3 к Договору, Заказчик уплачивает Исполнителю неустойку в размере \_\_\_\_\_(\_\_\_\_)[12]%, включая НДС, от суммы просроченного платежа за каждый календарный день просрочки, но не более \_\_\_\_\_(\_\_\_\_)[13] %, включая НДС, от этой суммы.

5.5. Обязательство Стороны по выплате неустойки возникает у нарушившей Стороны после получения ею письменного требования об уплате неустойки от другой Стороны.

5.6. Исполнитель не несёт ответственности за нарушение сроков выполнения работ, предусмотренных Договором, произошедших по вине Заказчика в следующих случаях:

5.6.1. несвоевременное выполнение согласованных Сторонами работ, возложенных на Заказчика и/или привлеченных им третьих лиц;

5.6.2. несвоевременное предоставление запрашиваемой Исполнителем в ходе выполнения работ информации, предоставление которой предусмотрено Договором;

5.6.3 несвоевременное оказание содействия Исполнителю, предусмотренного Договором (включая, предоставление технических ресурсов, рабочих мест, удовлетворяющих требованиям Исполнителя).

5.7. Исполнитель вправе не приступать к работе, а начатую работу приостановить в случаях нарушения Заказчиком своих обязательств по Договору, которые повлекли для Исполнителя невозможность выполнения работы, либо в случаях, предусмотренных п. п. 5.6 Договора. О приостановлении работ Исполнитель обязан незамедлительно письменно уведомить Заказчика, указав причины, по которым работы приостановлены и необходимые действия Заказчика по их устранению. Об устранении причин приостановления работ Заказчик обязуется незамедлительно письменно уведомить Исполнителя.

5.8. В каждом случае нарушения сроков исправления дефектов, классифицируемых как критические (в соответствии с Приложением № 5 к Договору), установленных в Приложении № 5 к

Договору, Исполнитель выплачивает Заказчику неустойку в размере \_\_\_\_\_ (\_\_\_\_\_) [14] %, включая НДС, от общей стоимости работ по Договору, за каждый \_\_\_\_\_ [15] просрочки, но не более \_\_\_\_\_ (\_\_\_\_\_) [16] % от этой суммы за каждый случай.

5.9. В каждом случае нарушения сроков исправления дефектов, классифицируемых как важные (в соответствии с Приложением № 5 к Договору), установленных в Приложении № 5 к Договору, Исполнитель выплачивает Заказчику неустойку в размере \_\_\_\_\_ (\_\_\_\_\_) [17] %, включая НДС, от общей стоимости работ по Договору, за каждый \_\_\_\_\_ [18] просрочки, но не более \_\_\_\_\_ (\_\_\_\_\_) [19] % от этой суммы за каждый случай.

5.10. В каждом случае нарушения сроков исправления дефектов, классифицируемых как средние (в соответствии с Приложением № 5 к Договору), установленных в Приложении № 5 к Договору, Исполнитель выплачивает Заказчику неустойку в размере \_\_\_\_\_ (\_\_\_\_\_) [20] %, включая НДС, от общей стоимости работ по Договору, за каждый \_\_\_\_\_ [21] просрочки, но не более \_\_\_\_\_ (\_\_\_\_\_) [22] % от этой суммы за каждый случай.

5.11. В каждом случае нарушения сроков исправления дефектов, классифицируемых как незначительных (в соответствии с Приложением № 5 к Договору), установленных в Приложении № 5 к Договору, Исполнитель выплачивает Заказчику неустойку в размере \_\_\_\_\_ (\_\_\_\_\_) [23] %, включая НДС, от общей стоимости работ по Договору, за каждый \_\_\_\_\_ [24] просрочки, но не более \_\_\_\_\_ (\_\_\_\_\_) [25] % от этой суммы за каждый случай.

5.12. Исполнитель несет ответственность за действия своих работников, выполняющих работы в помещениях Заказчика. Допуск работников Исполнителя к выполнению работ в рамках Договора в помещения Заказчика производится после их ознакомления уполномоченными лицами Заказчика с нормативными документами Заказчика по обеспечению информационной безопасности. После ознакомления с вышеуказанными документами, работники Исполнителя подписывают обязательство о выполнении требований этих документов.

5.13. В случае причинения Заказчику убытков в результате нарушения работниками Исполнителя требований нормативных документов Заказчика по обеспечению информационной безопасности в ходе выполнения работ в помещениях Заказчика, Исполнитель обязан полностью возместить Заказчику причиненные ему убытки.

## 6. ГАРАНТИЙНЫЕ ОБЯЗАТЕЛЬСТВА

6.1. Исполнителем производится гарантитное обслуживание ПО сроком \_\_\_\_\_ (\_\_\_\_\_) [26] со дня подписания Сторонами Акта сдачи-приемки работ.

6.2. В случае обнаружения дефектов в разработанном ПО в течение гарантийного срока, в соответствии с Классификацией дефектов (Приложение к Договору) (не включая дефекты в базовом программном обеспечении и аппаратных компонентах, принадлежащих Заказчику), Исполнитель обязуется исправлять их без дополнительной оплаты со стороны Заказчика, в сроки, указанные в Приложении № 5 к Договору. Срок исправления любого из дефектов отсчитывается от даты получения Исполнителем письменного уведомления от Заказчика об обнаруженном дефекте. Гарантия предоставляется на программное обеспечение и полнофункциональные скомпилированные исполняемые модули программ, которые были переданы Исполнителем Заказчику в рамках Договора и оформлены Актом сдачи-приемки работ. Если изменения в исходные тексты (коды) программ или модификации исполняемых модулей были внесены вне рамок Договора, либо эксплуатация ПО производится Заказчиком на программно-аппаратной платформе, не соответствующей ТЗ, либо неработоспособность ПО вызвана внесением Заказчиком изменений в базовое программное обеспечение, влияющее на работу ПО, то гарантия на такую версию ПО не распространяется.

6.3. Если в период гарантийного срока обнаружатся ошибки и дефекты ПО, которые не позволяют продолжить нормальную эксплуатацию ПО до их устранения, то гарантитный срок продлевается на период устранения таких ошибок и дефектов. Устранение ошибок и дефектов осуществляется Исполнителем своими силами и без дополнительной оплаты со стороны Заказчика.

6.4. Исполнитель гарантирует отсутствие в разработанном ПО скрытых (недокументированных) функциональных возможностей, ведущих к финансовому ущербу для Заказчика.

6.5. Техническая поддержка ПО по истечении гарантитного срока, указанного в п. 6.1. Договора, осуществляется Исполнителем на основании отдельно заключаемых Сторонами договоров, в которых определяются стоимость и порядок ее осуществления. В любом случае стоимость ежегодной технической поддержки не может превышать \_\_\_\_\_ (\_\_\_\_\_) [27] %, включая НДС, от стоимости разработанного ПО.

6.6. Если Заказчик сочтет необходимым, то Исполнитель соглашается с тем, что на полученное Заказчиком от Исполнителя ПО распространяются требования нормативного документа Заказчика, определяющего порядок проведения контрольной компиляции с исходных текстов ПО. После

ознакомления уполномоченного представителя Исполнителя с данным нормативным документом, он подписывает обязательство от лица Исполнителя о выполнении требований этого документа.

## **7. КОНФИДЕНЦИАЛЬНОСТЬ**

7.1. По взаимному согласию Сторон в рамках Договора конфиденциальной признается любая информация, касающаяся предмета Договора, хода его выполнения и полученных результатов. Каждая из Сторон обеспечивает защиту конфиденциальной информации, ставшей доступной ей в рамках Договора, от несанкционированного использования, распространения или публикации. Такая информация не будет передаваться третьим сторонам без письменного разрешения другой Стороны и использоваться в иных целях, кроме выполнения обязательств по Договору.

7.2. Любой ущерб, вызванный нарушением условий конфиденциальности, определяется и возмещается в соответствии с законодательством Российской Федерации.

7.3. Обязательства Сторон по защите конфиденциальной информации распространяются на все время действия Договора, а также в течение \_\_\_\_ (\_\_\_\_) [28] после прекращения действия Договора.

7.4. Не является нарушением режима конфиденциальности предоставление Сторонами информации по запросу уполномоченных государственных органов в соответствии с законодательством Российской Федерации.

## **8. ДЕЙСТВИЕ ДОГОВОРА**

8.1. Договор вступает в силу с момента подписания его Сторонами и действует до полного и надлежащего исполнения обязательств по нему.

8.2. Договор может быть расторгнут по соглашению Сторон. В любом случае досрочного расторжения Договора (за исключением случаев, указанных в п. п. 8.3. и 8.4. Договора) Стороны должны произвести между собой **взаиморасчеты** не позднее \_\_\_\_ (\_\_\_\_) [29] рабочих дней со дня расторжения, на основании двухстороннего акта, с учетом того, что Заказчику должна быть возвращена вся сумма полученного Исполнителем **аванса**, включая НДС, а Исполнителю оплачен результат фактически выполненной части работ, но только в том случае, если выполненная часть работ может быть использована Заказчиком в дальнейшем и Заказчик согласен принять выполненную часть работ, при этом решение о приемке / не приемке части работ Заказчик принимает исключительно по своему усмотрению, без согласования с Исполнителем.

8.3. Заказчик вправе в любой момент без объяснения причин расторгнуть Договор в одностороннем внесудебном порядке, письменно уведомив об этом Исполнителя не позднее чем за \_\_\_\_ (\_\_\_\_) [30] календарных дней до даты расторжения, указанной в уведомлении.

В этом случае Стороны должны произвести между собой взаиморасчеты не позднее \_\_\_\_ (\_\_\_\_) [31] рабочих дней со дня расторжения, на основании двухстороннего акта, с учетом того, что Заказчику должна быть возвращена вся сумма полученного Исполнителем аванса, включая НДС, а Исполнителю оплачен результат фактически выполненной части работ.

8.4. В случае расторжения Договора по инициативе Исполнителя (если такая инициатива не была вызвана ненадлежащим исполнением Заказчиком своих обязательств по Договору (в случаях, указанных в п. п. 5.4., 5.6. и 5.7. Договора) или обстоятельствами, предусмотренными Разделом 9 Соглашения) Заказчику в течение \_\_\_\_ (\_\_\_\_) рабочих дней со дня расторжения Договора должна быть возвращена вся сумма полученного Исполнителем аванса, включая НДС, и выплачена неустойка, включая НДС, в размере двойной ставки рефинансирования, установленной Банком России и действующей на день выплаты аванса, от суммы аванса за весь период, начиная с даты перечисления денежных средств Заказчиком по дату возврата аванса. Датой возврата аванса является дата зачисления денежных средств на счет Заказчика.

## **9. ФОРС-МАЖОР**

9.1 Любая из Сторон может быть освобождена от ответственности в определенных случаях, которые возникли независимо от ее воли.

9.2 Обстоятельства, вызванные не зависящими от воли Сторон событиями, которых добросовестная Сторона не могла избежать или последствия которых она не могла устранить, считаются случаями, которые освобождают от ответственности, если они произошли после заключения Договора и препятствуют его полному или частичному исполнению.

9.3 Случаями непреодолимой силы считаются следующие события: война, военные действия, массовые беспорядки, забастовки, эпидемии, природные катастрофы, а также акты органов власти, влияющие на выполнение обязательств Сторон, и все другие аналогичные события и обстоятельства.

9.4 Сторона, пострадавшая от действия непреодолимой силы, обязана известить другую Сторону заказным письмом или иным доступным ей способом сразу же после наступления форс-мажорных

обстоятельств и разъяснить, какие меры необходимы для их устранения, но в любом случае не позднее \_\_\_\_\_ (\_\_\_\_\_) [32] календарных дней после начала действия непреодолимой силы.

9.5 Несвоевременное уведомление об обстоятельствах непреодолимой силы лишает соответствующую Сторону права на освобождение от ответственности по причине указанных обстоятельств. Обстоятельства непреодолимой силы должны быть подтверждены документально компетентными органами.

9.6 Если указанные обстоятельства продолжаются более \_\_\_\_\_ (\_\_\_\_\_) [33] месяцев, каждая Сторона имеет право инициировать досрочное расторжение настоящего Договора. В этом случае, Стороны производят взаиморасчеты на основании двустороннего Акта, подписанного Сторонами.

## **10. ПРАВА ИНТЕЛЛЕКТУАЛЬНОЙ СОБСТВЕННОСТИ**

10.1. Исключительное право на результат работ (разработанное ПО) в полном объеме без ограничений принадлежит Заказчику с момента подписания Сторонами Акта сдачи-приемки работ.

10.2. Исполнитель гарантирует, что при разработке ПО по Соглашению не будут нарушены авторские, смежные и любые иные права третьих лиц.

В случае если к Заказчику будут предъявлены претензии (требования, иски) со стороны третьих лиц по поводу нарушения их прав в результате использования Заказчиком полученного от Исполнителя ПО, Исполнитель по получении извещения от Заказчика обязуется выступить на стороне Заказчика, оказать всемерное содействие Заказчику при урегулировании таких претензий, в том числе взять на себя обязанность по подготовке и проведению досудебных переговоров и переписки с такими третьими лицами, а впоследствии (в том случае если Заказчик будет вынужден в силу вступившего в силу решения суда или если по согласованию с Исполнителем будет признано приемлемым возместить ущерб третьих лиц во внесудебном порядке) возместить Заказчику в полном объеме выплаченные Заказчиком третьим лицам денежные средства, все связанные с нарушением прав третьих лиц судебные издержки Заказчика и иные расходы, а также уплатить Заказчику штраф в размере \_\_\_\_\_ (\_\_\_\_\_) [34] %, включая НДС, от подлежащей возмещению суммы.

Возмещение и выплата штрафа производится Исполнителем не позднее \_\_\_\_\_ (\_\_\_\_\_) [35] рабочих дней со дня получения соответствующего требования от Заказчика.

Если по решению суда Заказчик не может пользоваться ПО или иным результатом услуг, или в случае если Исполнитель и/или Заказчик желает прекратить текущее использование ПО из соображений устранения нарушения прав третьего лица, Исполнитель обязан незамедлительно без дополнительной оплаты со стороны Заказчика заменить программное обеспечение, являющееся предметом претензий третьих лиц, таким образом, чтобы права третьих лиц не нарушались.

## **11. УВЕДОМЛЕНИЯ**

11.1. Все уведомления, извещения и сообщения в связи с выполнением Договора должны быть оформлены в письменном виде на **русском языке** и могут быть направлены с помощью средств факсимильной связи, электронной почтой, заказной или курьерской почтой, с подтверждением факта их получения, по фактическим адресам Сторон, приведенным в Статье 15 Договора, либо по адресу, указанному ниже для соответствующей Стороны, либо по иному адресу, о котором любая из Сторон может уведомить другую Сторону.

11.2. Информация считается полученной Сторонами:

11.2.1 в случае направления с помощью средств факсимильной связи или по электронной почте – в дату, указанную в подтверждении о получении Стороной-получателем факсимильного сообщения или сообщения электронной почты, имеющемся у Стороны-отправителя;

11.2.2 в случае направления заказной или курьерской почтой – на дату, указанную в подтверждении о вручении отправления Стороне-получателю, имеющемуся у Стороны-отправителя.

11.3. Исполнитель обязуется регулярно представлять Заказчику отчет о ходе выполнения работ по Договору в формате ведения учета у Заказчика.

## **12 ПОРЯДОК РАССМОТРЕНИЯ СПОРОВ**

12.1. Все споры между Сторонами, возникшие в ходе исполнения Договора, подлежат рассмотрению в Арбитражном суде г. Москвы.

## **13 ЗАКЛЮЧИТЕЛЬНЫЕ ПОЛОЖЕНИЯ**

13.1. При изменении адреса, реквизитов или уполномоченных (ответственных) лиц Сторон данная Сторона обязуется уведомить об этом другую Сторону незамедлительно, но в любом случае не позднее \_\_\_\_\_ (\_\_\_\_\_) [38] календарных дней. До получения Стороной уведомления о таких изменениях исполнение Договора этой Стороной, совершённое с использованием имеющихся у неё сведений, считается надлежащим.

13.2. Договор составлен в \_\_\_\_ (\_\_\_\_)[39] экземплярах, которые подписываются обеими Сторонами и имеют одинаковую юридическую силу, \_\_\_\_ (\_\_\_\_) экземпляр \_\_\_\_ (\_\_\_\_) – для Исполнителя и \_\_\_\_ (\_\_\_\_) – для Заказчика.

## 14 СПИСОК ПРИЛОЖЕНИЙ

- 14.1. Приложение. Техническое задание.
- 14.2. Приложение. План-график выполнения работ.
- 14.3. Приложение. График платежей.
- 14.4. Приложение. Форма Акта сдачи-приемки работ.
- 14.5. Приложение. Классификация дефектов и сроки их устранения.
- 14.6. Приложение. Форма Протокола несоответствий.
- 14.7. Приложение. Форма Дополнительного Соглашения.

## 15 АДРЕСА, РЕКВИЗИТЫ И ПОДПИСИ СТОРОН

Исполнитель:

Местонахождение: \_\_\_\_\_  
Тел.: : \_\_\_\_\_, факс: : \_\_\_\_\_  
БИК : \_\_\_\_\_,  
Счет : \_\_\_\_\_,  
Кор/счет : \_\_\_\_\_  
в : \_\_\_\_\_,  
ОКПО : \_\_\_\_\_, ОКВЭД : \_\_\_\_\_,  
КПП : \_\_\_\_\_, ИНН : \_\_\_\_\_,  
ОГРН : \_\_\_\_\_

От Исполнителя:

Должность \_\_\_\_\_  
ФИО \_\_\_\_\_  
подпись  
М. П.

Заказчик:

Местонахождение: : \_\_\_\_\_  
Тел.: : \_\_\_\_\_, факс: : \_\_\_\_\_  
БИК : \_\_\_\_\_,  
Счет : \_\_\_\_\_,  
Кор/счет : \_\_\_\_\_  
в : \_\_\_\_\_,  
ОКПО : \_\_\_\_\_, ОКВЭД : \_\_\_\_\_,  
КПП : \_\_\_\_\_, ИНН : \_\_\_\_\_,  
ОГРН : \_\_\_\_\_

От Заказчика:

Должность \_\_\_\_\_  
ФИО \_\_\_\_\_  
подпись  
М. П.

### Приложение 5

к Договору № \_\_\_\_\_  
от " \_\_\_\_ " \_\_\_\_\_ 20 \_\_\_\_ г.

Классификация дефектов и сроки их устраниния  
Классификация дефектов

Критические	Определение
K_1	Дефект, вызывающий повреждение или разрушение операционной системы.
K_2	Дефект, вызывающий повреждение структуры базы данных или потерю данных в определенных таблицах.
K_3	Дефект, делающий невозможным дальнейшую работу или запуск программы.
K_4	Дефект, вызывающий зависание программы или компьютера, а также вызывающий критическую ошибку ОС.
K_5	Дефект, после проявления которого, невозможно дальнейшее использование какой-либо функциональности.
K_6	Не реализованная функциональность.
K_7	Дефект, вызывающий нарушение информационной безопасности
Важные	
B_1	Дефект, проявляющийся только после определенной последовательности действий, после проявления которого, затруднено дальнейшее использование какой-либо функциональности.

B_2	Дефект, проявляющийся часто, не имеющий четкой последовательности действий к нему приводящей, не вызывающий эффектов, описанных в К_1 и К_4, после проявления которого, затруднено дальнейшее использование какой-либо функциональности.
B_3	Дефект, вызывающий непредвиденное использование ресурсов, не указанных в Техническом задании.
B_4	Искаженный внешний вид пользовательского интерфейса в Web проектах при использовании версий Web-browser'ов, указанных в Техническом задании.
Средние	
C_1	Появление неправильных сообщений или отсутствие требуемых.
C_2	Искаженный внешний вид пользовательского интерфейса, который затрудняет работу пользователя, но оставляет возможность работы с программой.
C_3	Дефект, проявляющийся редко, не имеющий четкой последовательности действий к нему приводящей, не вызывающий эффектов, описанных в К_1 и К_4.
Незначительные	
H_1	Искаженный внешний вид пользовательского интерфейса, который затрудняет работу пользователя. Для примера – ошибки правописания, неточная прокрутка и т. д. Для Web проектов такие дефекты классифицируются по B_4.
H_2	После деинсталляции программы остаются файлы и записи в реестре или конфигурационных файлах.
H_3	Другие дефекты.

- [1] Указать краткую характеристику программного обеспечения (например: прикладного, системного и т. п.)
- [2] Указать название программного обеспечения
- [3] Перечислить составляющие компонент (например: модули, технические задания, документация и другие артефакты проектирования).
- [4] Указать вид носителя
- [5] Указать цифрами и прописью количество экземпляров
- [6] Указать цифрами и прописью количество возвращаемых экземпляров.
- [7] Указать цифрами и прописью
- [8] Указать цифрами и прописью
- [9] Указать цифрами и прописью
- [10] Указать проценты цифрами и прописью (размер неустойки должен быть экономически обоснован и стимулировать Исполнителя на надлежащее исполнение обязательств по Договору)
- [11] Указать проценты цифрами и прописью, но не менее 10 %
- [12] Указать проценты цифрами и прописью (размер неустойки должен быть экономически обоснован и стимулировать Исполнителя на надлежащее исполнение обязательств по Договору)
- [13] Указать проценты цифрами и прописью, но не более 10 %
- [14] Указать проценты цифрами и прописью (размер неустойки должен быть экономически обоснован и стимулировать Исполнителя на надлежащее исполнение обязательств по Договору)
- [15] Указать временной интервал (например: день (календарный, рабочий), час)
- [16] Указать проценты цифрами и прописью, но не менее 10 %
- [17] Указать проценты цифрами и прописью (размер неустойки должен быть экономически обоснован и стимулировать Исполнителя на надлежащее исполнение обязательств по Договору)
- [18] Указать временной интервал (например: день (календарный, рабочий), час)
- [19] Указать проценты цифрами и прописью, но не менее 10 %
- [20] Указать проценты цифрами и прописью (размер неустойки должен быть экономически обоснован и стимулировать Исполнителя на надлежащее исполнение обязательств по договору)
- [21] Указать временной интервал (например: день (календарный, рабочий), час)
- [22] Указать проценты цифрами и прописью
- [23] Указать проценты цифрами и прописью (размер неустойки должен быть экономически обоснован и стимулировать Исполнителя на надлежащее исполнение обязательств по договору)

- [24] Указать временной интервал (например: день (календарный, рабочий), час)
- [25] Указать проценты цифрами и прописью
- [26] Указать цифрами и прописью (срок не должен быть менее 12 месяцев).
- [27] Указать проценты цифрами и прописью, но не более чем 20 %
- [28] Указать срок цифрами и прописью. Срок должен составлять не менее 5 лет.
- [29] Указать цифрами и прописью
- [30] Указать цифрами и прописью
- [31] Указать цифрами и прописью
- [32] Указать цифрами и прописью
- [33] Указать цифрами и прописью
- [34] Указать проценты цифрами и прописью (размер штрафа должен быть экономически обоснован)
- [35] Указать цифрами и прописью
- [36] Указать ФИО, тел., факс, e-mail представителя (представителей) Заказчика
- [37] Указать ФИО, тел., факс, e-mail представителя (представителей) Исполнителя
- [38] Указать цифрами и прописью
- [39] Указать цифрами и прописью (для всего пункта)
- [40] Указать в строках столбца номера этапов (если работы выполняются без разбиения на этапы, столбец не заполняется)
  - [41] Перечислить в строках столбца выполняемые на этапе работы (если работы выполняются без разбиения на этапы – указать: «работы по разработке программного обеспечения «\_\_\_\_\_» (указать наименование ПО)»
  - [42] Указать в строках столбца даты начала работ
  - [43] Указать в строках столбца даты завершения работ
  - [44] Указать в строках столбца наименования документов, служащих основанием для завершения работ (например: Акт сдачи-приемки работ, Акт сдачи-приемки работ по этапу (заключительному этапу))
  - [45] Указать цифрами и прописью количество экземпляров
  - [46] Указать входящие в комплект: техническую документацию и других материалы и документы, предусмотренные ТЗ
  - [47] Указать цифрами и прописью количество возвращаемых экземпляров
  - [48] Указать цифрами и прописью
  - [49] Указать цифрами и прописью
  - [50] Указать в столбце наименование платежа (например: аванс, оплата работ по этапу, окончательный расчет и т. д.)
  - [51] Указать в столбце дату платежа
  - [52] Указать в столбцах суммы (сумма аванса за выполняемые работы не может превышать 50 % от стоимости работ)
  - [53] Указать основание для проведения платежа (например: счет на авансовый платеж, Акт сдачи-приемки работ, Акт сдачи-приемки работ по этапу (заключительному этапу)). Основанием для авансового платежа является счет, выставленный Исполнителем, в остальных случаях (закрытие этапа, сдача-приемка выполненных работ) – соответствующий Акт, подписанный Сторонами.
  - [54] Для Акта сдачи-приемки работ по промежуточному этапу данный абзац исключить
  - [55] Указать в каком объеме (например: полном, неполном; в последнем случае в строке «претензии Заказчика к выполненным работам» указать конкретно, что выполнено, а что – нет)
  - [56] Указать в какие сроки (например: установленные, досрочно, с просрочкой; в последнем случае в строке «претензии Заказчика к выполненным работам» указать, с какой именно (количество дней))
  - [57] Указать с каким качеством (например: надлежащим, ненадлежащим; в последнем случае в строке «претензии Заказчика к выполненным работам» указать, в чём именно выражается нарушение обязательств по качеству)
  - [58] Указать конкретные претензии Заказчика
  - [59] Указать срок цифрами и интервал времени (дни, часы) – применить ко всему столбцу

## **ЛАБОРАТОРНАЯ РАБОТА № 2**

### **Разработка UML модели проекта в Visual Studio 2013. Использование Visual Studio 2013**

**Цель работы:** получить навык создания на основе UML модели в системе **Visual Studio 2013** и изучить особенности создания на основе UML кода на языке C# .

**Продолжительность работы – 4 часа.**

#### **Содержание**

1. Модель жизненного цикла программы .....	25
2. Моделирование структуры программы на языке UML .....	26
3. Пример генерации кода.....	31
4. Пример доработки кода .....	36
5. Разработка и реорганизация кода: рефакторинг.....	40
6. Порядок выполнения лабораторной работы.....	43
7. Вопросы .....	43

#### **1. Модель жизненного цикла программы.**

Одна из наиболее важных возможностей интегрированной среды Visual Studio 2013 – поддержка всех этапов жизненного цикла разработки программного проекта.

Концепция жизненного цикла разработки состоит из следующих этапов:

*Выработка требований и целей (Для больших проектов – этап разработки архитектуры системы) – формулировка технических, маркетинговых, эксплуатационных и других требований будущей программной системы;*

*Спецификация – формализованное, полное, точное и внешнее описание программной системы (термин внешнее в данном случае понимается описание того, "ЧТО, а не КАК", т.е. элементы и проект реализации в спецификацию не входят);*

*Проектирование (дизайн) – разработка подробного проекта системы, включая иерархию модулей, входные и выходные данные, информационные потоки, представление данных, основные алгоритмы; частью спецификации и проектирования является моделирование – построение формальной модели проекта;*

*Реализация (кодирование) – разработка программного кода системы на выбранном для реализации языке программирования;*

*Верификация – проверка корректности реализации программной системы, которая на практике в большинстве случаев выполняется путем тестирования (прогона набора тестов), либо путем формальной верификации – формального доказательства того, что реализация системы соответствует формальной спецификации, выполненной на каком-либо языке спецификаций.*

*Выпуск (релиз) программной системы для пользователей;*

*Сопровождение программной системы – исправление ошибок, обучение пользователей, ответы на их вопросы, реализация расширений функциональности системы по требованию пользователей.*

## 2. Моделирование структуры программы на языке UML

В Visual Studio 2013, как и во многих других современных интегрированных средах, имеется поддержка моделирования структуры программы на языке UML. Данный язык моделирования может быть использован на ранних этапах разработки проекта.

Язык UML позволяет спроектировать иерархию классов в абстрактных терминах, представить ее в виде модели, а модель – в виде диаграммы. Затем по диаграмме может быть сгенерирован код на выбранном языке (например, на C#). Таким образом, использование языка UML позволяет перейти от этапа моделирования и проектирования к этапу реализации. Сгенерированный код может затем использоваться как основа для последующей реализации проекта.

Однако автоматически сгенерированный код не является законченной программой. В коде достаточно подробно описана структура данных, иерархия классов и параметры методов класса. Но методы представляются в виде заглушек и требуют доработки до работающей программы.

Тем не менее, такая возможность позволяет сэкономить время, затрачиваемое на описание классов, что уже само по себе не так уж и мало.

В качестве примера рассмотрим генерацию простой UML-модели, визуализацию этой модели и генерацию по ней кода на языке C#. Для создания UML-моделей используется специальный вид проекта Modeling Project.

Создадим проект для моделирования. В среде VS 2013 выберем в главном меню File / NewProject и вид проекта Modeling Project (Рис. 1).

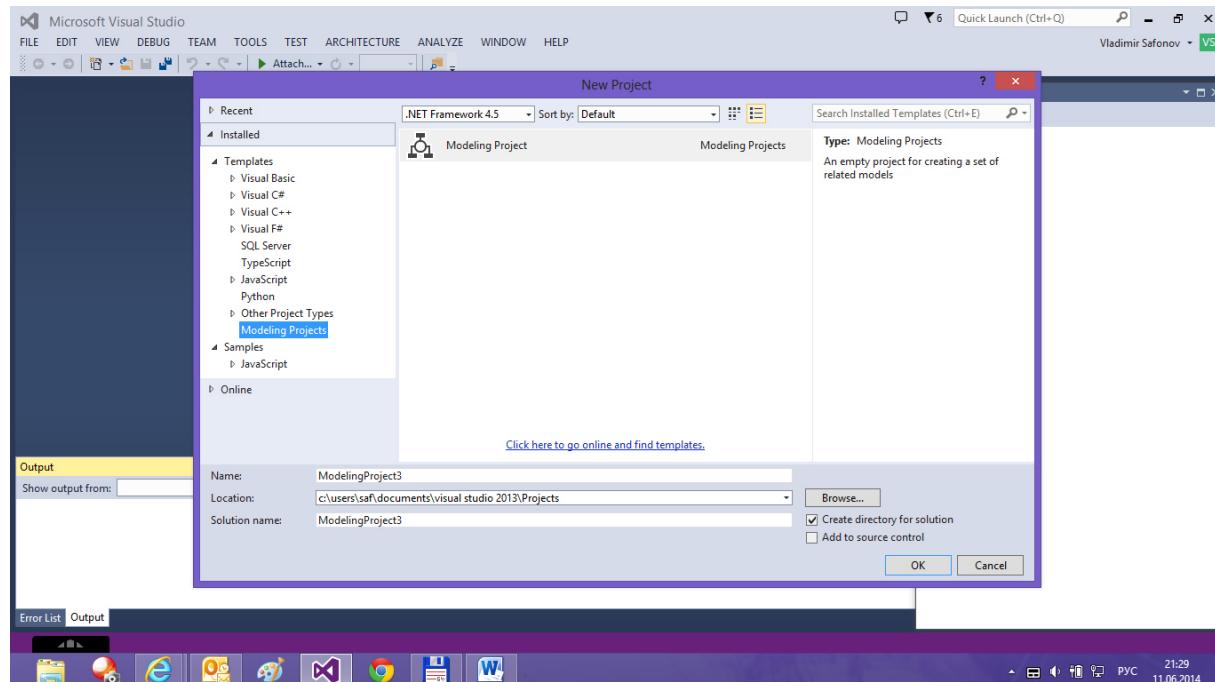


Рис. 1. Создание проекта для построения UML-модели

Для добавления к проекту новой UML-диаграммы выберем пункт меню Architecture, который специально предназначен для отображения архитектуры программы в виде UML-модели, и в нем пункт Add New Diagram (Рис. 2).

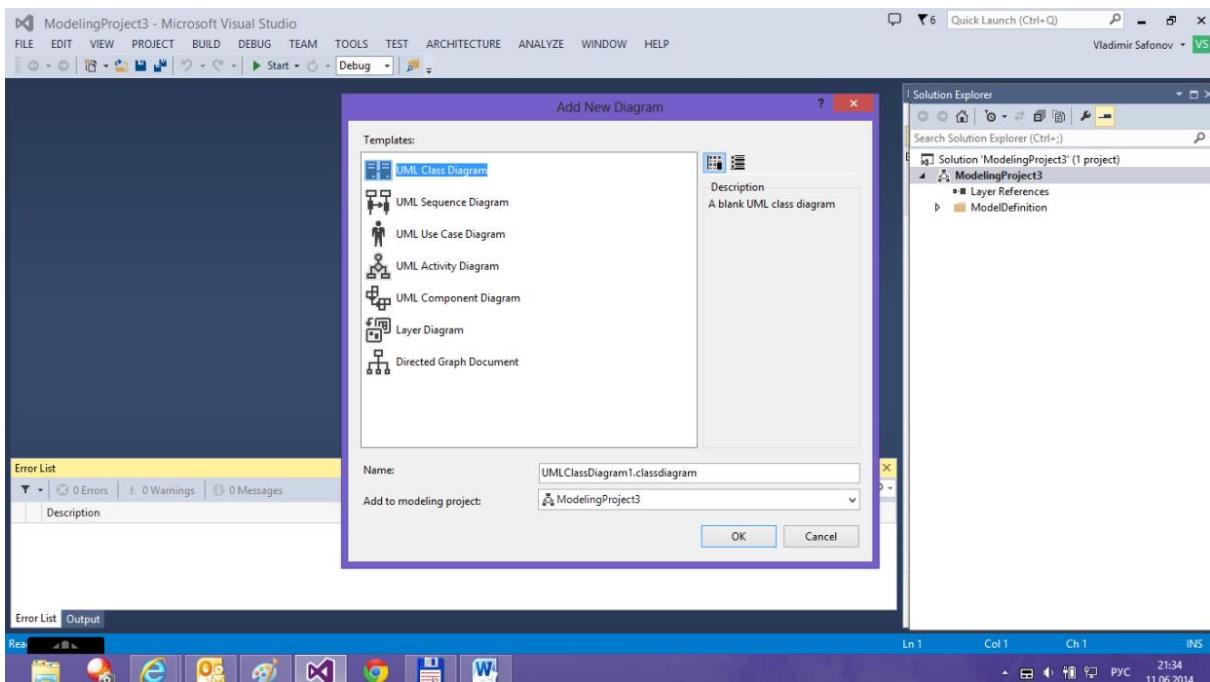


Рис. 2. Выбор вида новой UML-диаграммы

По умолчанию это новая диаграмма классов (UML Class Diagram), как наиболее часто используемая. Однако в этом пункте меню могут быть созданы и другие виды диаграмм:

UML Sequence Diagram – диаграмма последовательности;

UML Use Case Diagram – диаграмма использования;

UML Activity Diagram – диаграмма активности;

UML Component Diagram – диаграмма компонент;

Layer Diagram – диаграмма уровней;

Directed Graph Document – диаграмма, изображающая документ в виде ориентированного графа.

Создадим диаграмму использования (Use case) (Рис. 3, 4).

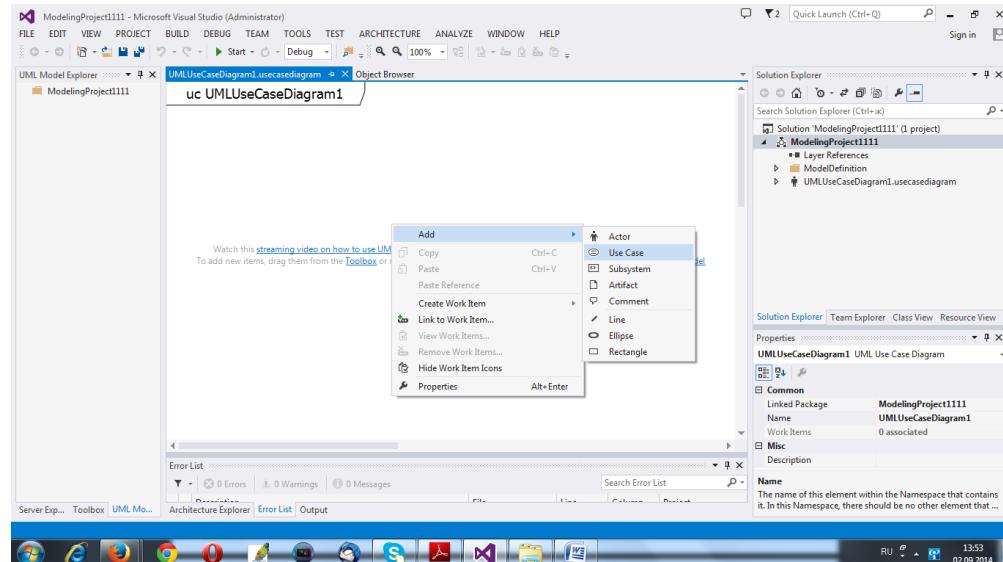


Рис. 3. Создание новой диаграммы использования (Use Case Diagram)

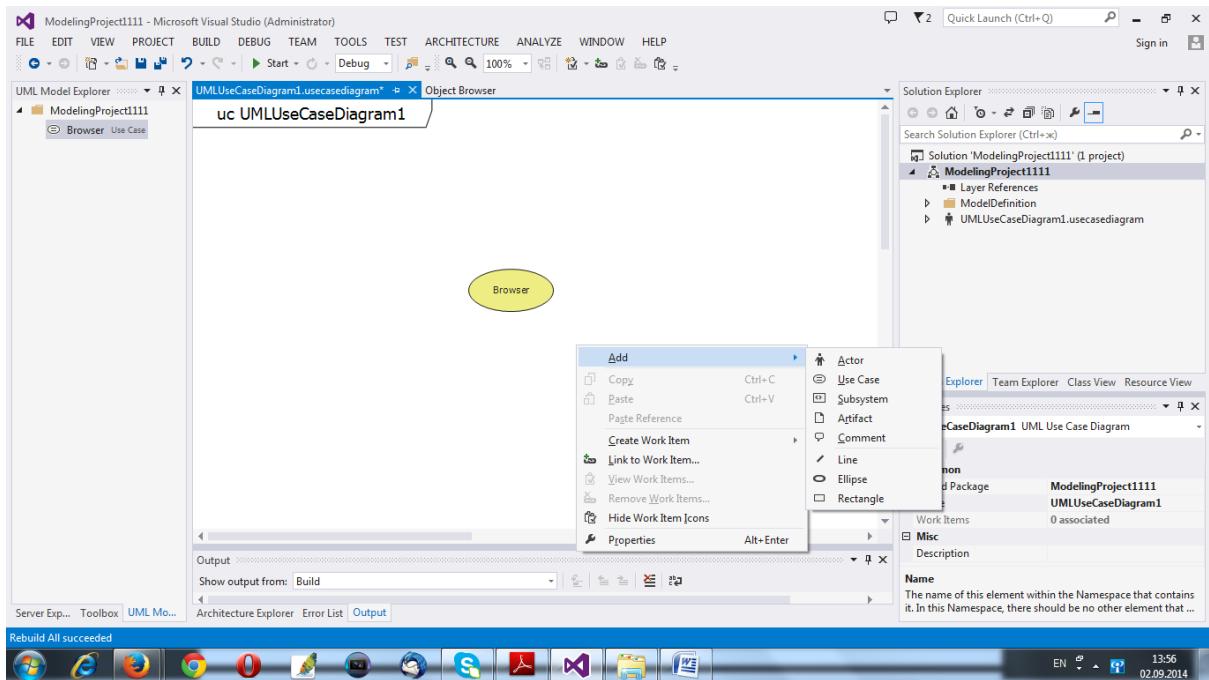


Рис. 4. Создание новой диаграммы использования (Use Case Diagram)

Создадим актеров, прецеденты, комментарии и установим связи между элементами (Рис. 5).

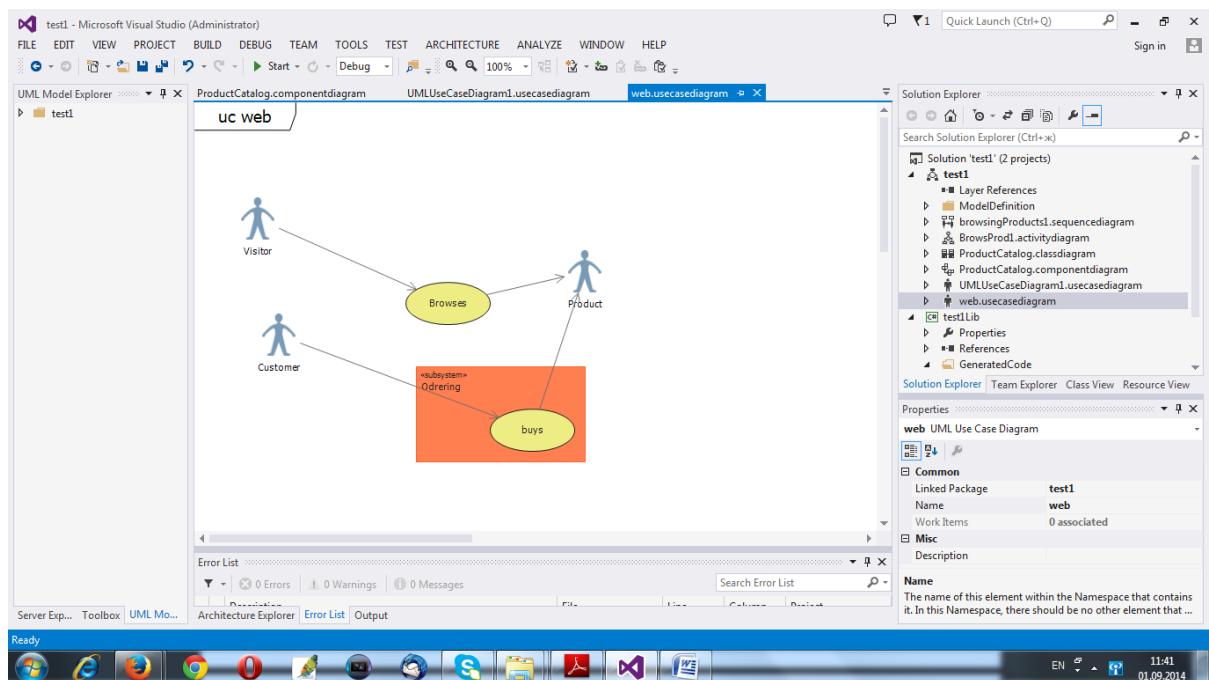


Рис. 5. Создание актеров, прецедентов, комментариев и связей между элементами

Построим диаграмму активностей (Activity Diagram) (Рис 6, 6а, 6б).

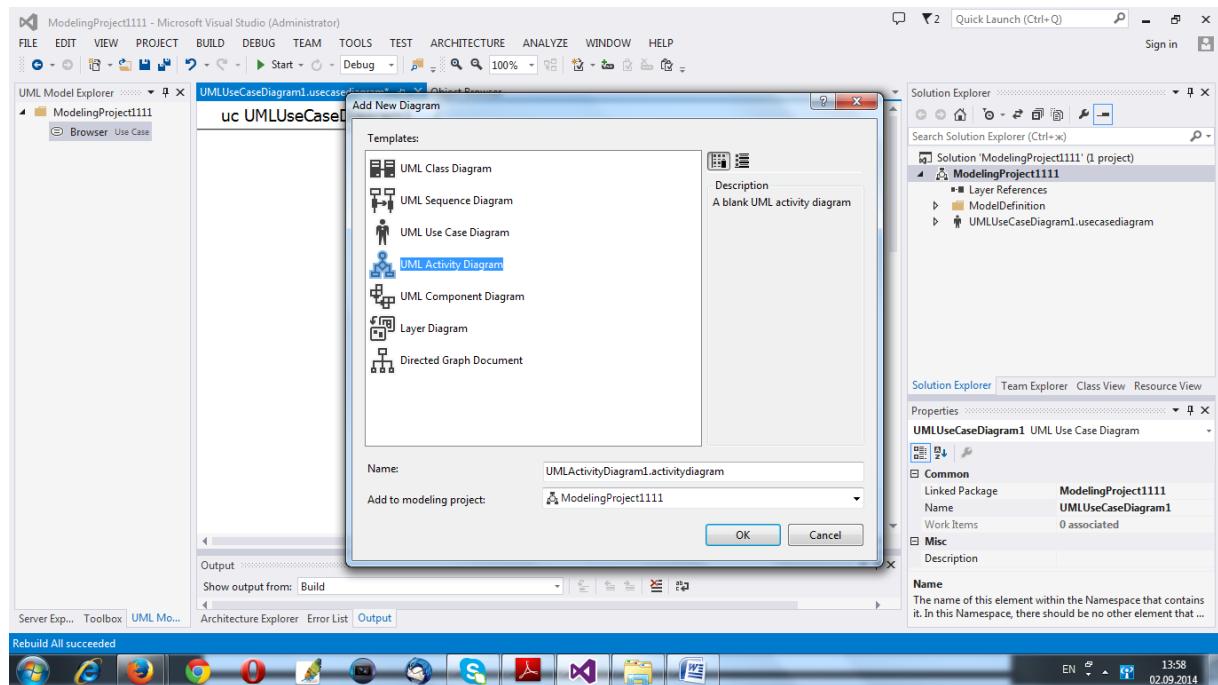


Рис. 6. Построение диаграммы активностей Activity Diagram.

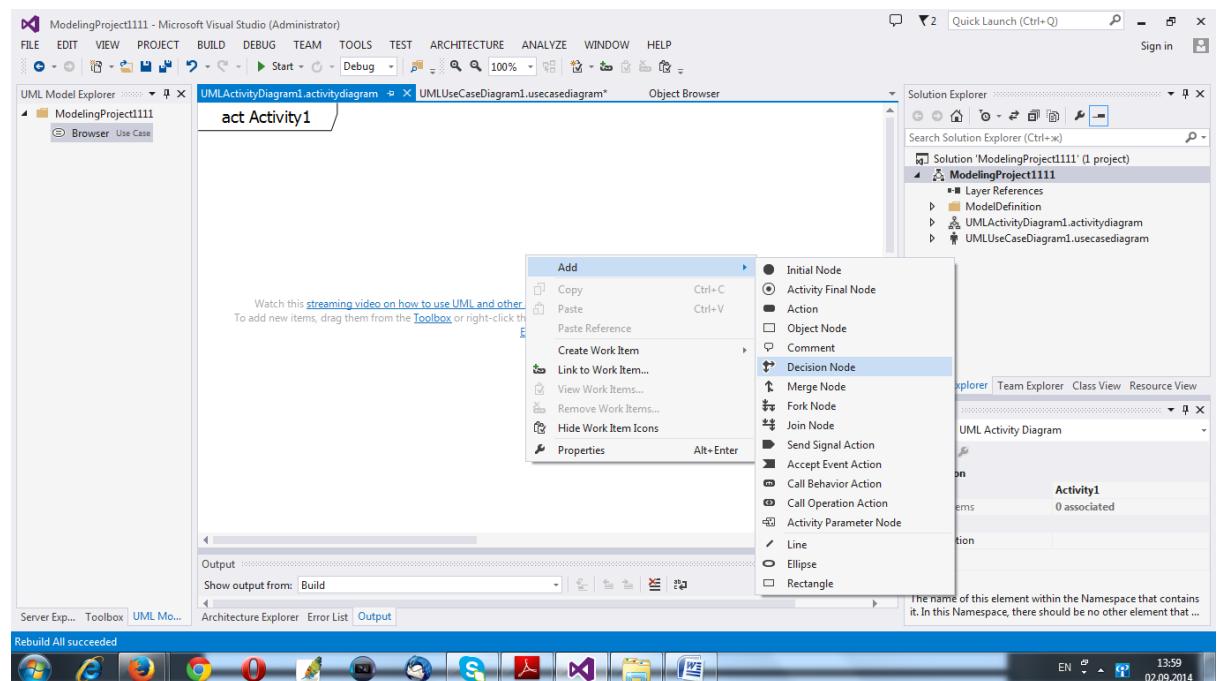


Рис. 6а. Выбор элементов диаграммы активностей.

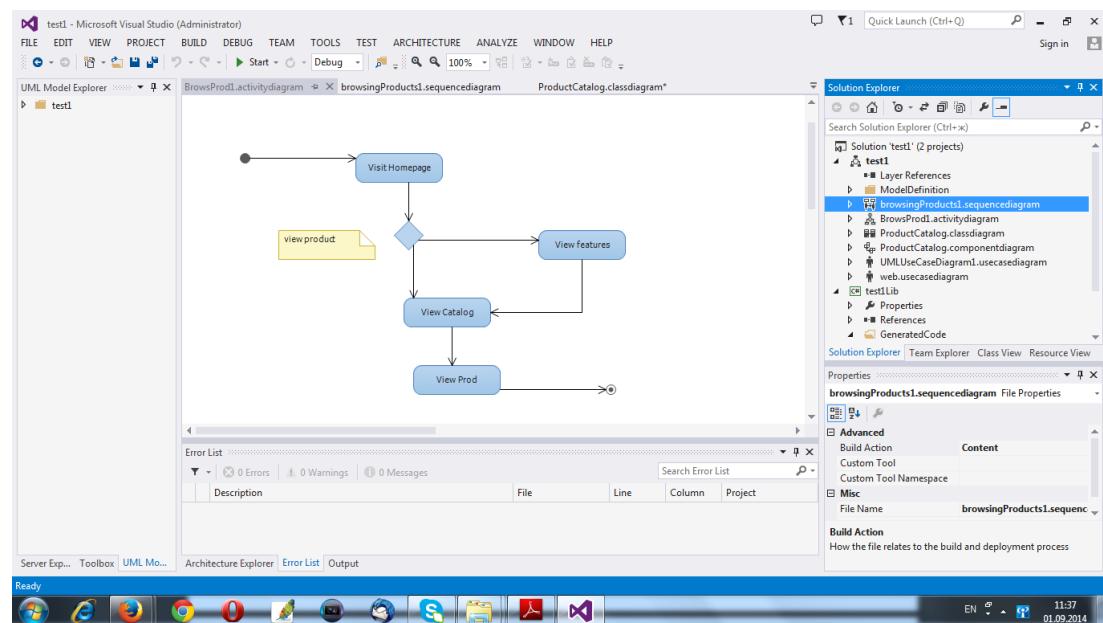
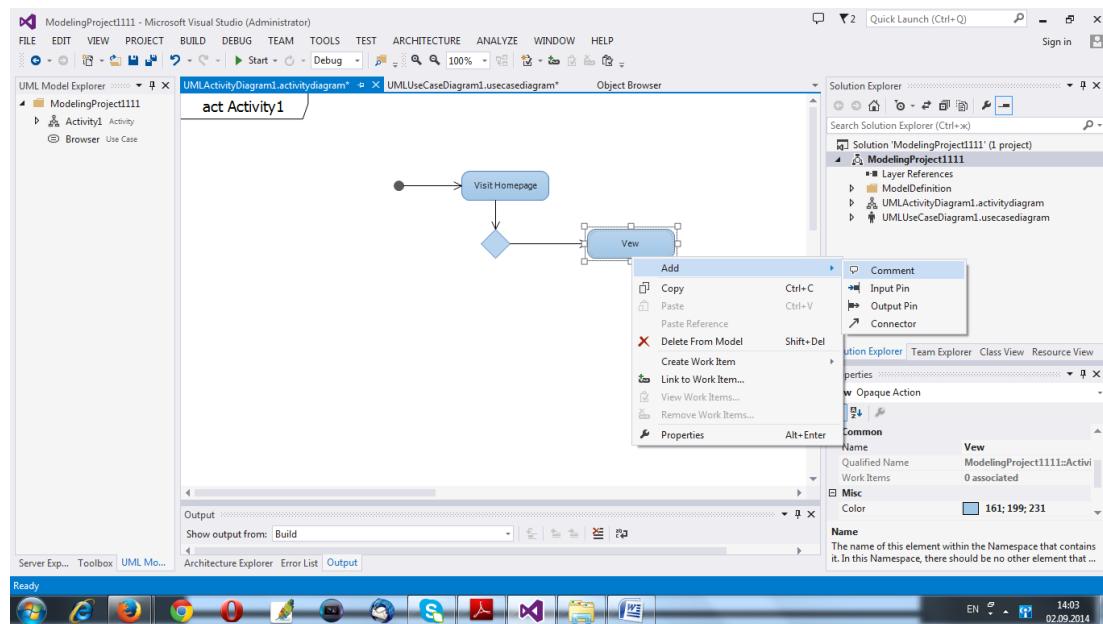


Рис. 66. Диаграмма активностей.

## Построение диаграммы последовательностей (Sequence Diagram) (Рис. 7.).

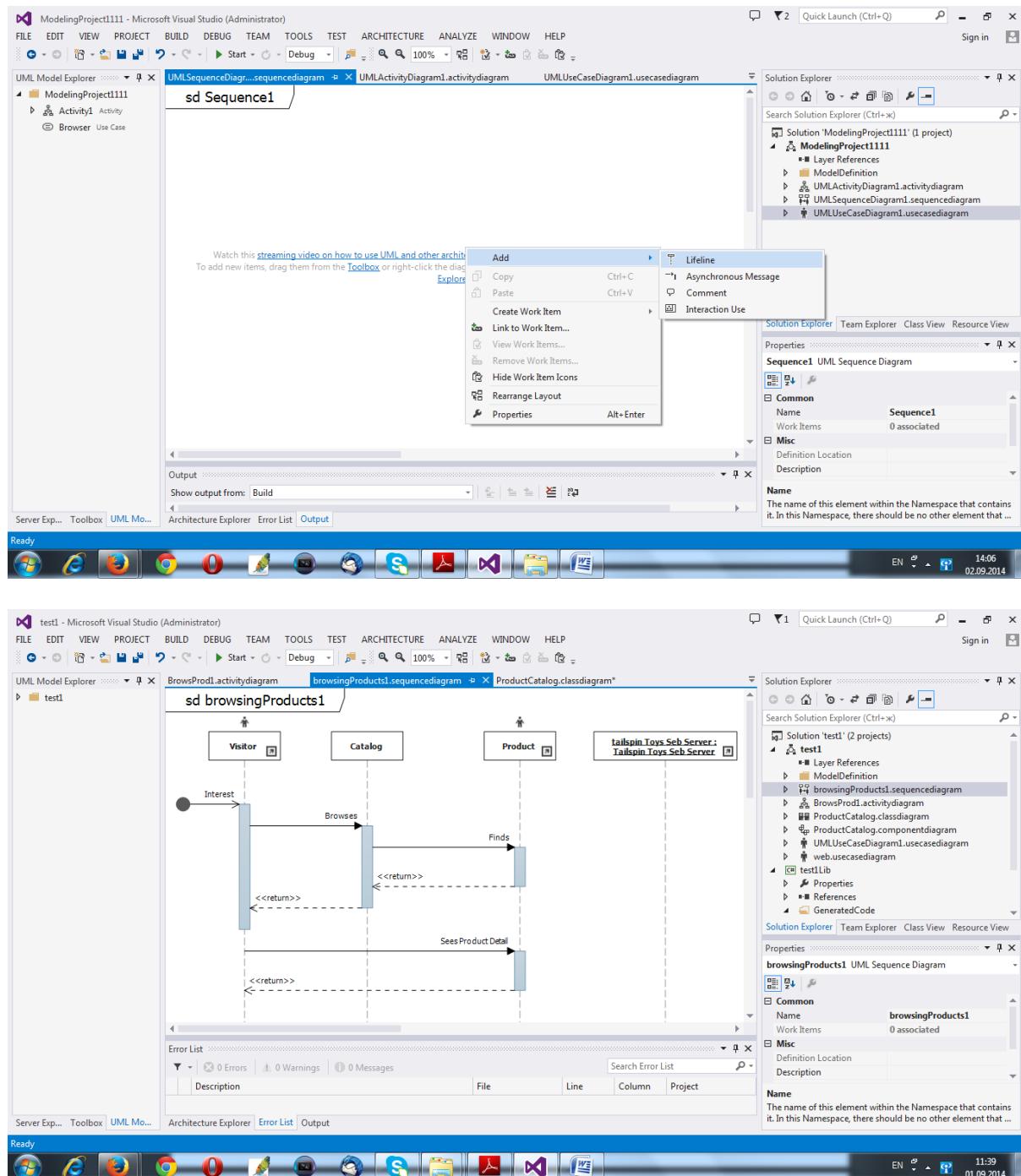


Рис. 7. Построение диаграммы последовательностей (Sequence Diagram).

### 3. Пример генерации кода из диаграммы UML

В качестве примера создадим UML диаграмму для построения простой консольной программы преобразования строковых переменных.

Перейдем к созданию описания некоторой персоны, включающее в себя имя, отчество и фамилию. Далее имитируем ввод этих данных пользователем и произведем

отображение этих данных на консоль, затем проведем операцию преобразования имени и отчества в инициалы и снова отобразим эту информацию на консоль.

Создаем новую диаграмму классов, пока – пустую (Рис. 8):

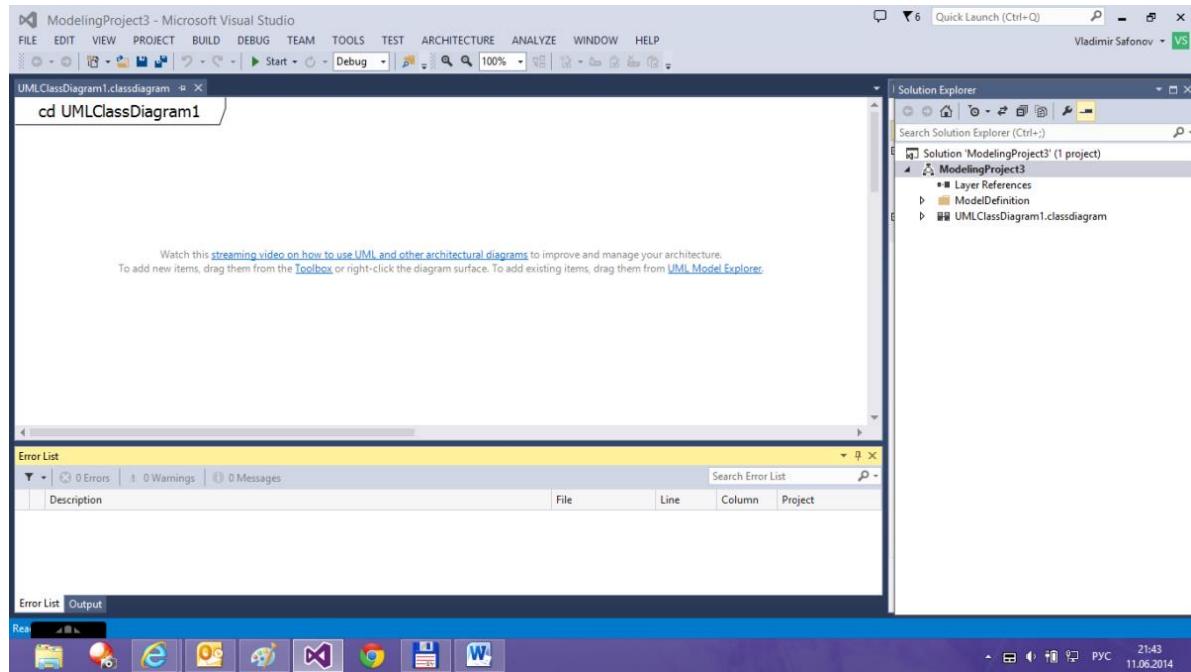


Рис. 8. Создание новой диаграммы классов

Для добавления к диаграмме класса выбираем в контекстном меню пункты Add / Class (Рис. 9):

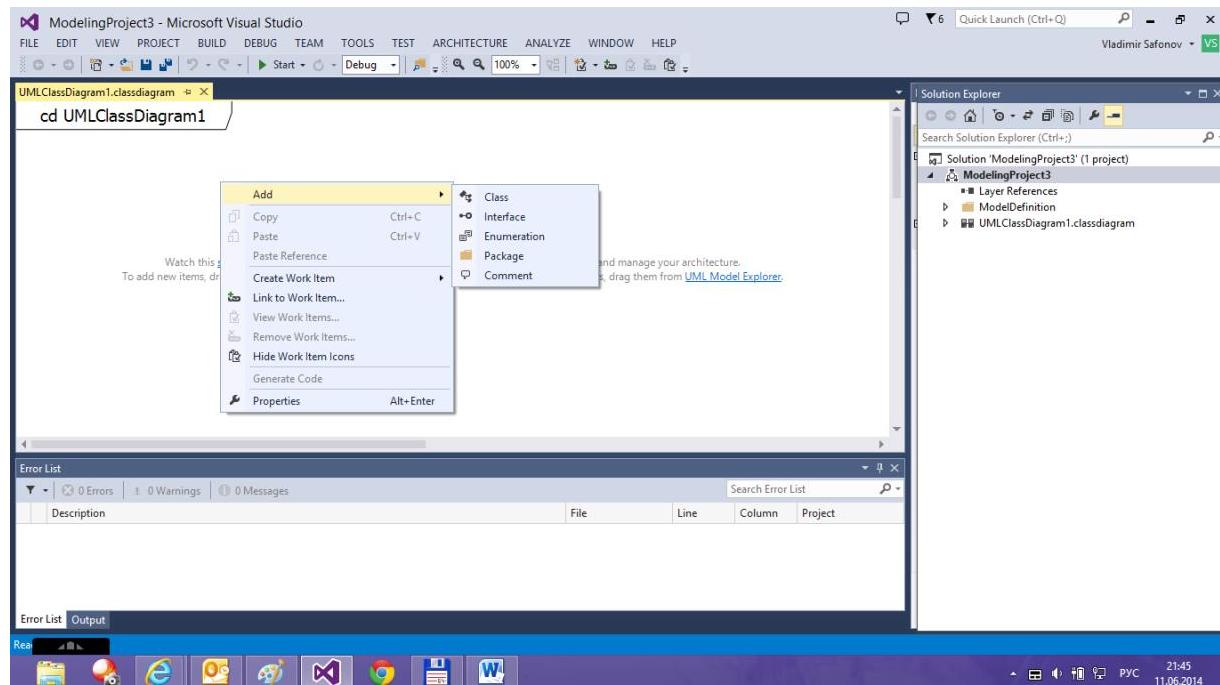


Рис. 9. Добавление к диаграмме нового класса

Класс добавляется с именем Class1 по умолчанию. По терминологии UML, класс состоит из атрибутов и операций (см. рис. 10).

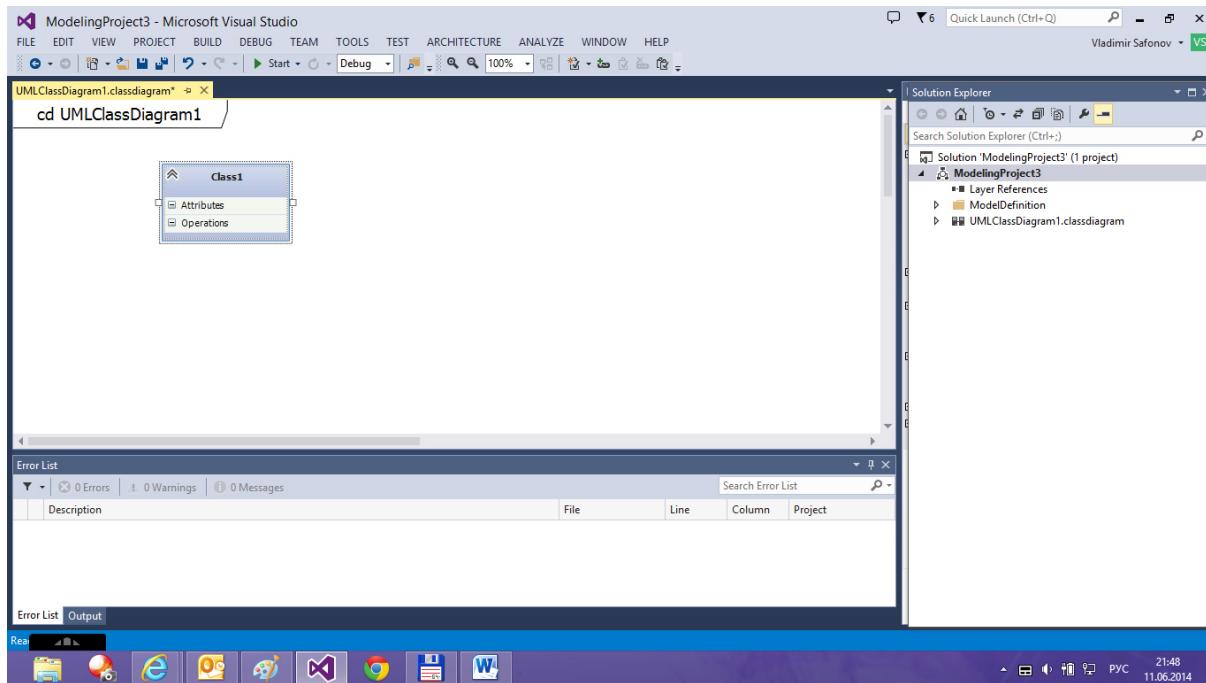


Рис. 10. Класс с атрибутами и операциями

Создадим два класса: Programm и Person. Класс Programm содержит описание основной программы, Person – описание объекта класса и методов класса. (Рис. 11.)

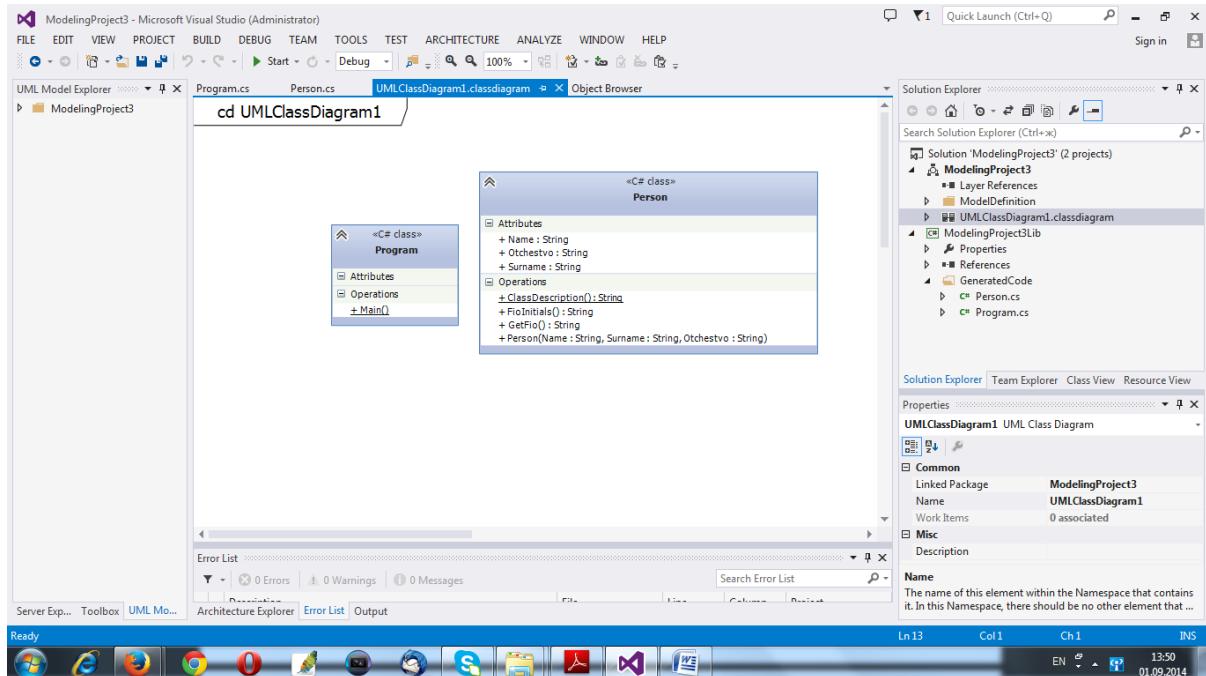


Рис. 11. Создаем два класса с атрибутами и операциями.

Так как цель лабораторной работы носит ознакомительный характер с возможностями Visual Studio, мы не будем усложнять себе задачу созданием большого количества классов и установлением различных видов связи между ними, хотя такая возможность существует (Рис. 12).

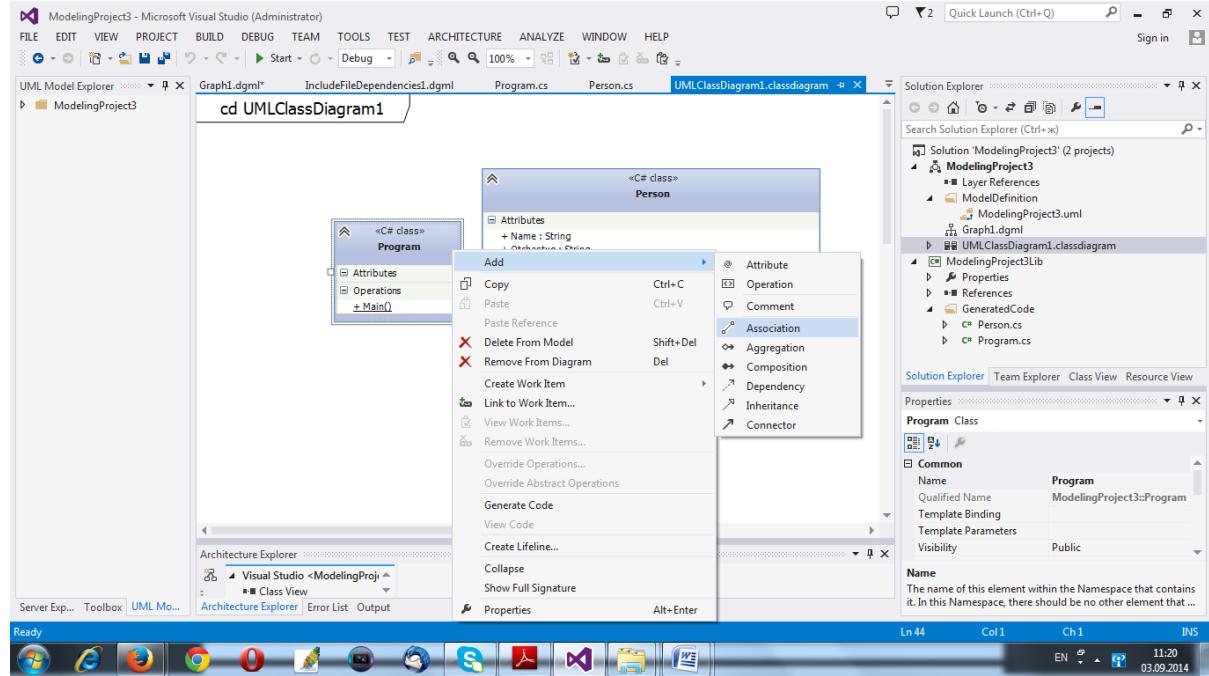


Рис. 12. Контекстное меню класса.

Теперь можно сгенерировать код, который используем при дальнейшей разработке. В контекстном меню выбираем пункт Generate code (Рис. 13):

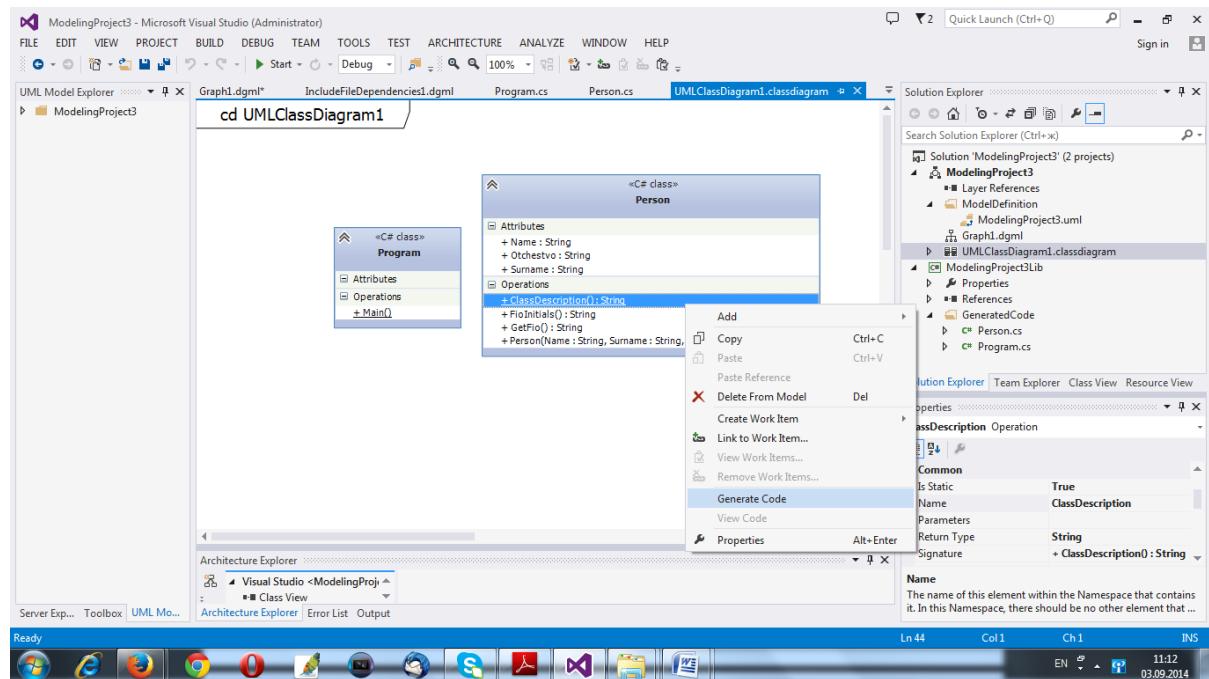


Рис. 13. Выбор операции создания кода.

Генератор кода просит уточнить, по какому шаблону будет происходить генерация. Выбираем шаблон кода для класса (Рис. 14):

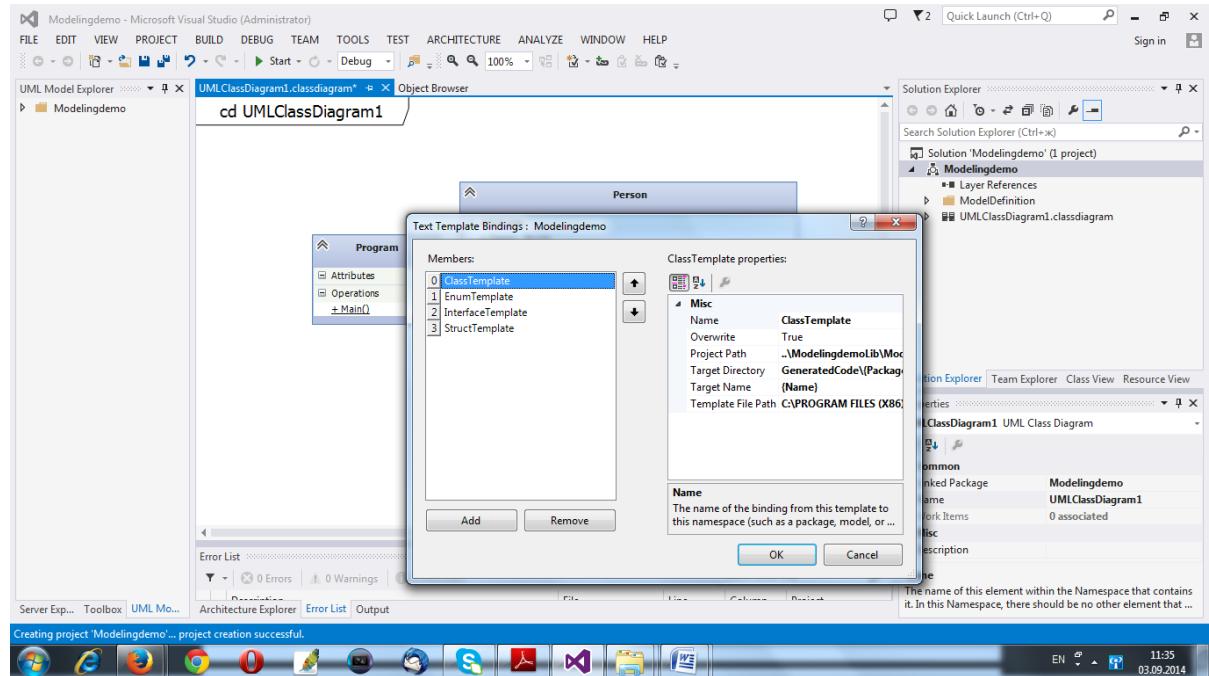


Рис. 14. Выбор шаблона кода для класса при генерации кода по UML-диаграмме

После генерации кода в Solution Explorer появляются два новых пункта – файлы на языке C# Person.cs и Program.cs (Рис. 15):

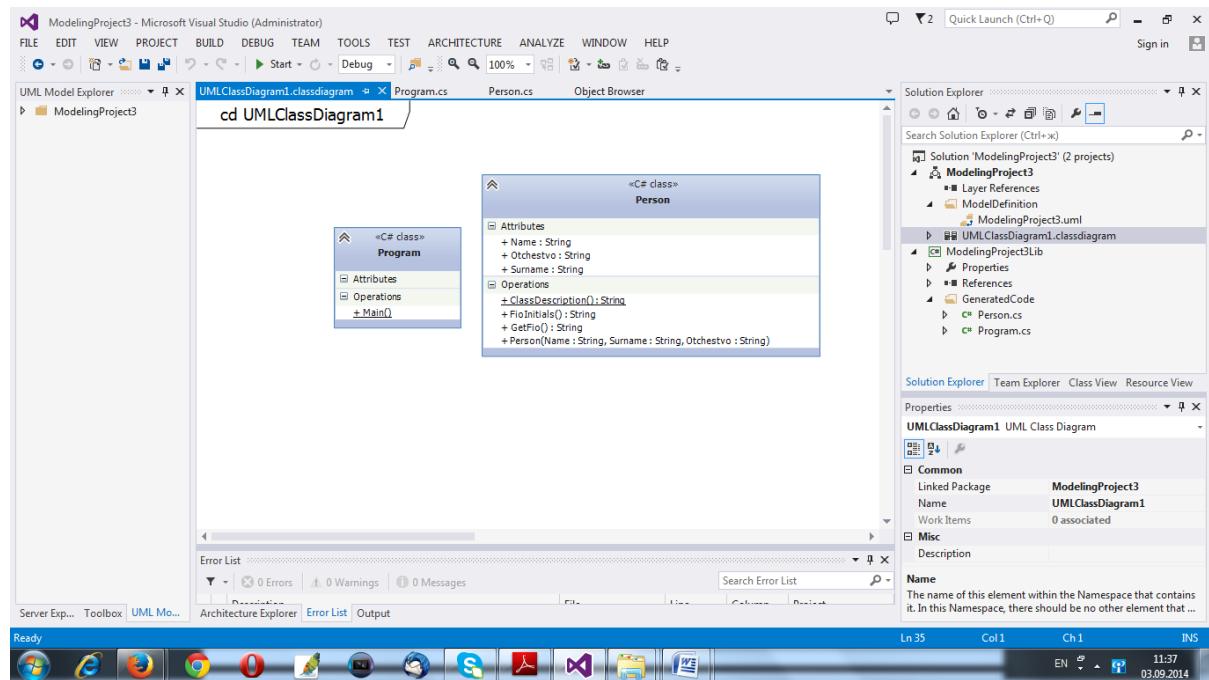


Рис. 15. Завершение генерации кода по UML-диаграмме: генерация двух файлов на C#

Фрагмент файла Person.cs показан на рис. 16.

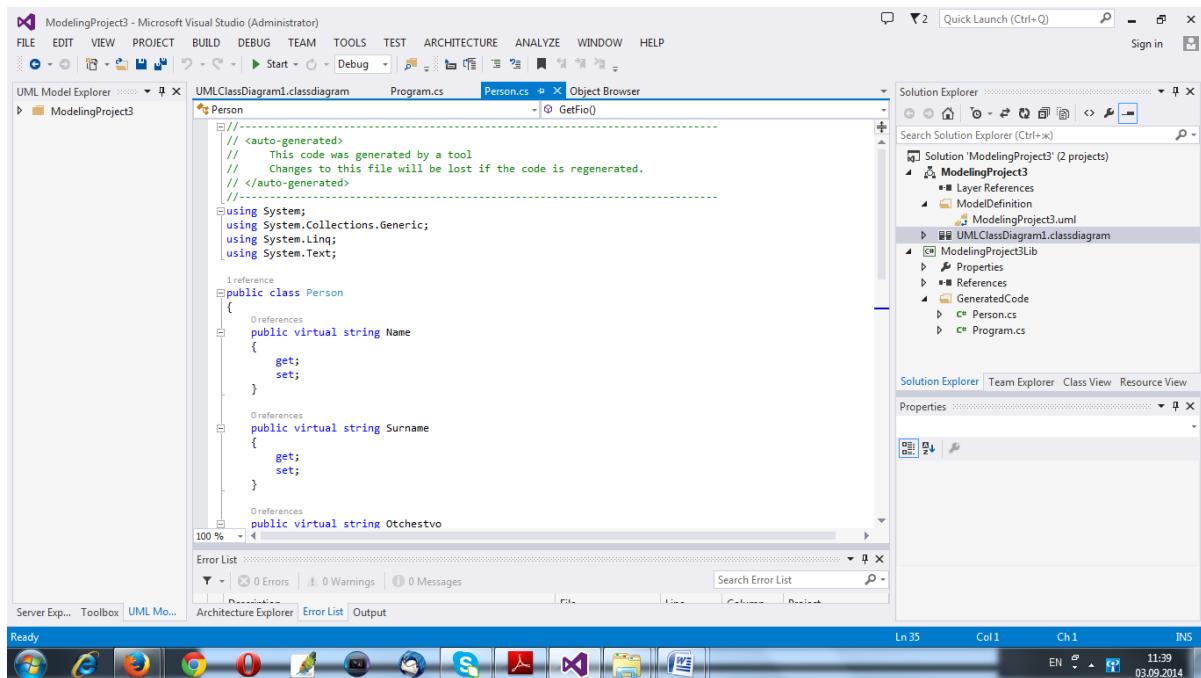


Рис. 16. Сгенерированный файл Person.cs

Атрибуты класса реализованы в виде свойства (property) с методами get и set.

Заглушки методов – в виде виртуальных методов с реализацией в виде генерации исключения, например:

```
public virtual string GetFio()  
{  
    throw new System.NotImplementedException();  
}
```

Теперь сгенерированные файлы можно использовать при последующей разработке.

Созданная модель играет весьма важную роль: это отражение в проекте результата раннего этапа разработки – моделирования и проектирования. При необходимости изменить проект, изменения могут быть сделаны в UML-диаграммах, по которым исходные коды генерируются автоматически.

#### 4. Пример доработки кода

Автоматически сгенерированный код имеет вид:

```
//-----  
//<auto-generated>  
// This code was generated by a tool  
// Changes to this file will be lost if the code is regenerated.  
//</auto-generated>  
//-----  
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;
```

```

public class Person
{
    public virtual string Name
    {
        get;
        set;
    }

    public virtual string Surname
    {
        get;
        set;
    }

    public virtual string Otchestvo
    {
        get;
        set;
    }

    public virtual string GetFio()
    {
        throw new System.NotImplementedException();
    }

    public virtual string FioInitials()
    {
        throw new System.NotImplementedException();
    }

    public static string ClassDescription()
    {
        throw new System.NotImplementedException();
    }

    public Person(string Name, string Surname, string Otchestvo)
    {
    }
}

//-----
//<auto-generated>
// This code was generated by a tool
// Changes to this file will be lost if the code is regenerated.
//</auto-generated>
//-----
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

public class Program
{
    public static void Main()

```

```

    {
        throw new System.NotImplementedException();
    }
}

```

В автоматически созданной структуре необходимо уточнить описание атрибутов и методов:

```

//-----
//<auto-generated>
// This code was generated by a tool
// Changes to this file will be lost if the code is regenerated.
//</auto-generated>
//-----

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

public class Person
{
    private string _name = "";
    public virtual string Name
    {
        get{ return _name; }
        set { _name = value; }
    }

    private string _surname = "";

    public virtual string Surname
    {
        get { return _surname; }
        set { _surname = value; }
    }

    private string _otchestvo = "";
    public virtual string Otchestvo
    {
        get { return _otchestvo; }
        set { _otchestvo = value; }
    }

    public virtual string Fio
    {
        get
        {
            string fio = Surname + " " + Name + " " + Otchestvo;
            return fio;
        }
    }

    public virtual string FioInitials
    {
        get
        {
            string fio = Surname + " " + Name.Substring(0, 1) + ". " + Otchestvo.Substring(0, 1) + ".";
        }
    }
}

```

```

        return fio;
    }
}

public static string ClassDescription
{
    get
    {
        return "Класс Person. Хранит данные о человеке.";
    }
}

public Person(string name, string surname, string otchestvo)
{
    Name = name;
    Surname = surname;
    Otchestvo = otchestvo;
}

}

//-----
//<auto-generated>
// This code was generated by a tool
// Changes to this file will be lost if the code is regenerated.
//</auto-generated>
//-----

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

public class Program
{
    public static void Main(string[] args)
    {
        // создаем два экземпляра класса человек с разными фио
        Person person1 = new Person("Пушкин", "Александр", "Сергеевич");
        Person person2 = new Person("Гончарова", "Наталья", "Николаевна");

        System.Console.WriteLine(person1.Fio);
        System.Console.WriteLine(person1.FiolInitials);
        System.Console.WriteLine(person2.Fio);
        System.Console.WriteLine(person2.FiolInitials);
        System.Console.WriteLine(Person.ClassDescription);
        System.Console.WriteLine(Person.ClassDescription);
    }
}

```

Как видим, задача не сводится к чисто механическим действиям и требует серьезных навыков программирования. И, тем не менее, автоматическая генерация кода снимает с программиста часть рутинных процессов чем существенно ускоряет процесс программирования.

## 5. Разработка и реорганизация кода: рефакторинг

Особое место при разработке и модификации кода занимает рефакторинг – систематическая модификация и улучшение существующего кода, без коренного изменения его семантики, с помощью автоматических преобразований, осуществляемых средой. Другими словами, это способ приведения кода в порядок, при котором шансы появления новых ошибок в коде минимальны.

Изменения при рефакторинге кода вносятся пошагово, мелкими операциями. Переименовать функцию, разбить ее на несколько подфункций, изменить сигнатуру метода – все эти операции легко выполнить так, чтобы не занести в код ошибок. Однако совокупный эффект ряда таких элементарных операций может быть весьма значительным – от упрощения структуры кода, до полного изменения архитектуры разрабатываемой программы.

Между тем, проведение рефакторинга вручную – достаточно утомительное занятие. Например, чтобы изменить порядок следования параметров в методе класса, мало изменить сигнатуру метода – необходимо пройти по всему коду программы и модифицировать все вызовы данного метода. Изменение тривиальное, но требует активного "копипастинга".

Среда Visual Studio впервые обзавелась такими средствами в версии VS2005. Разработчики, использующие C# и J#, получили в свое распоряжение следующий набор инструментов:

- Rename – переименование имени переменной, метода, класса и т.п. с автоматическим обновлением всех ссылок на это имя в коде;
- Extract method – оформление выделенной части кода в новый, отдельный метод;
- Encapsulate field – создание свойства, скрывающего выбранную переменную-член класса;
- Extract interface – создание интерфейса на основе списка методов класса;
- Promote local variable to parameter – вынесение локальной переменной в параметр метода;
- Remove/Reorder parameters – удаление параметров метода и изменение порядка их следования с автоматическим обновлением всех ссылок в коде на данный метод.

В качестве примера применим рефакторинг для нашей программы- изменим имя метода. Задача состоит в том, чтобы изменить имя метода и в его определении, и во всех его использованиях. В общем случае, если проект достаточно велик, вручную выполнять решение подобной задачи весьма неудобно. Почти наверняка при этом какие-либо использования метода будут забыты, и к ним придется возвращаться уже после получения ошибок при компиляции (сборке) проекта.

Применим рефакторинг к созданной нами программе (Рис. 17):

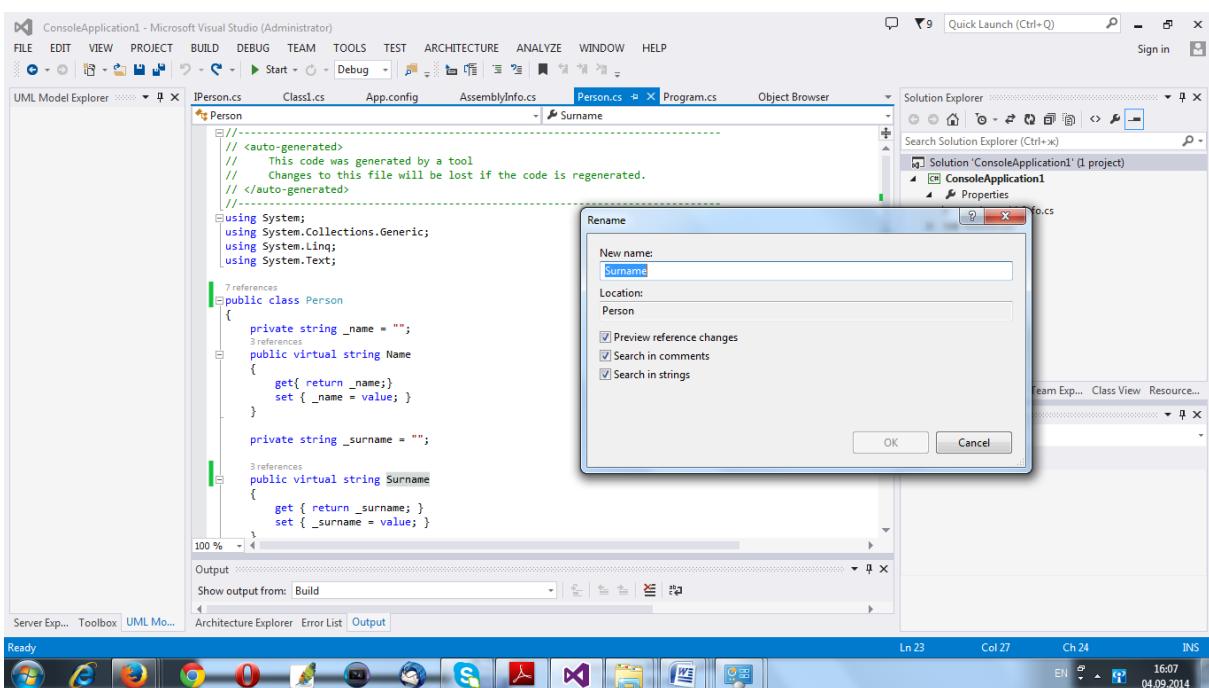
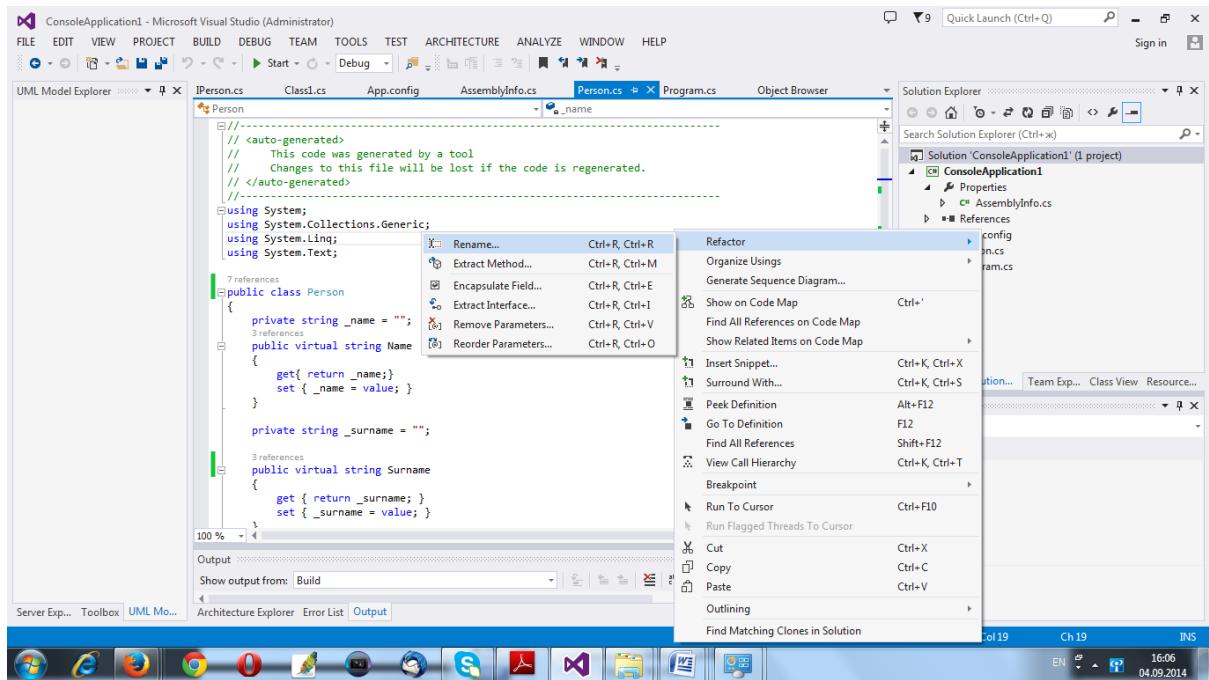


Рис. 17. Рефакторинг имени

В качестве примера изменим имя Surname на Surname1. В окне Preview Changes отобразятся все замены имени по тексту (Рис. 18):

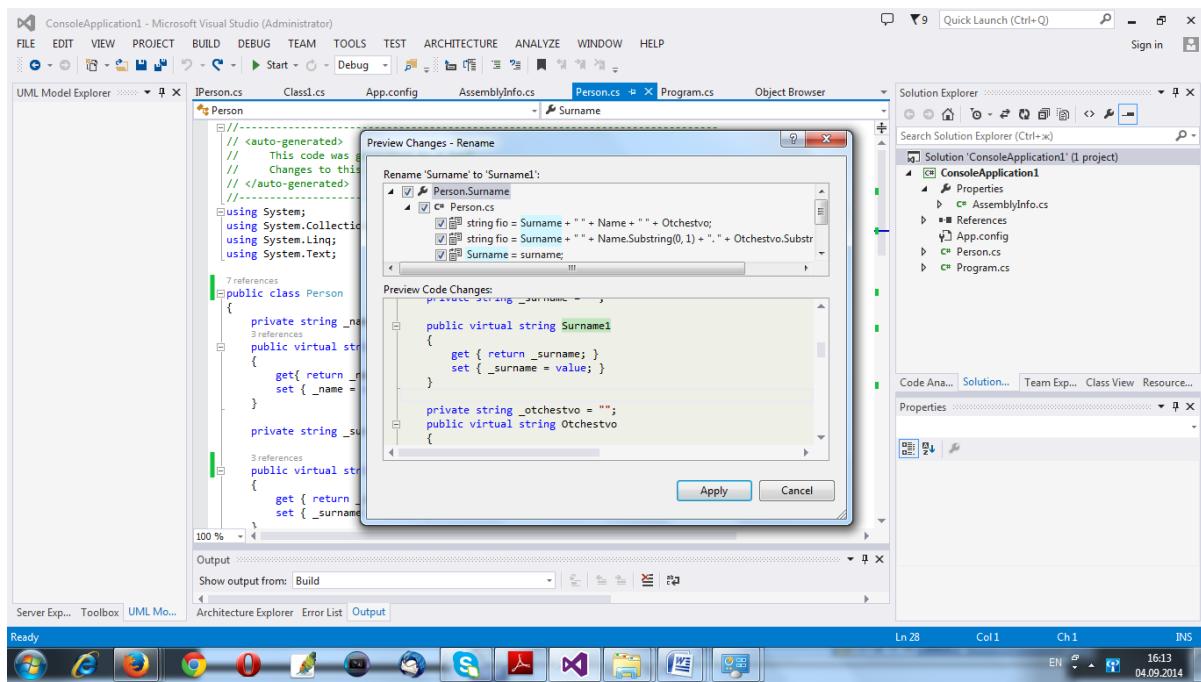


Рис. 18. Отображение изменений в тексте программы

Поскольку все предлагаемые изменения нас устраивают, нажимаем Apply.  
На рис. 19 показан результат рефакторинга.

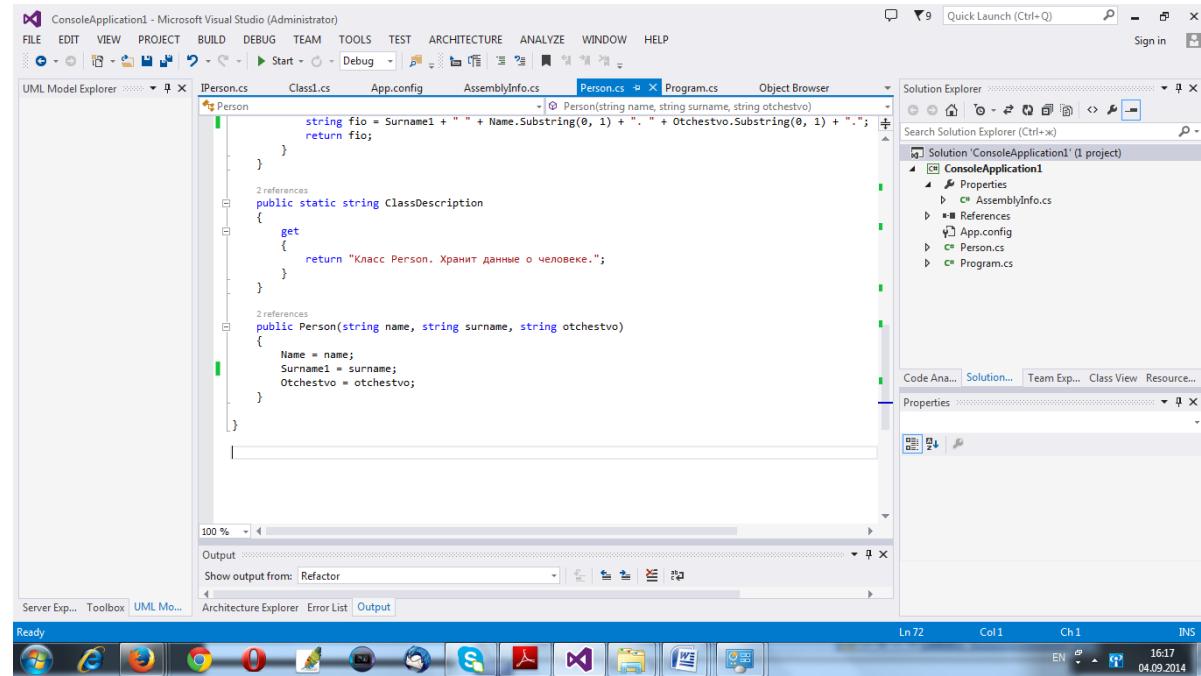


Рис. 19. Результат рефакторинга – имя метода изменено

Чтобы убедиться в правильности выполненных средой преобразований, выполним запуск программы.

Реализованный набор инструментов, конечно же, упростил жизнь разработчикам, однако он далеко не идеален. Из недостатков можно отметить:

- отсутствие поддержки C++ и VB;
- достаточно медленная работа;
- набор средств рефакторинга минимален, ряд полезных инструментов в нем отсутствует;
- интерфейс далеко не самый эргономичный и требует активного использования мышки.

## **7. Порядок выполнения лабораторной работы.**

1. Изучить теоретические сведения, приведенные в описании лабораторной работы, письменно ответить на контрольные вопросы, создать и выполнить рассмотренный в лабораторной работе пример.
2. Запустить систему Microsoft Visual Studio 2013.
3. Создать новый проект: выбрать команду главного меню File | New | Project, Modeling Project. Далее набрать имя проекта Lab1Project в поле Name, с помощью кнопки Browse указать в поле Location папку на сетевом диске Н для хранения проекта и нажать кнопку OK. Для добавления к проекту новой UML-диаграммы выберем пункт меню Architecture и вид создаваемой диаграммы.
4. Далее в соответствии с индивидуальным заданием (Лабораторная работа №1) создать Use-case диаграмму, диаграмму классов, диаграмму последовательностей, диаграмму использования, диаграмму активностей.
5. По созданной диаграмме классов произвести генерацию кода. Доработать код до работающей программы, используя текст из описания лабораторной работы. Сохранить на сетевом диске Н проект приложения командой главного меню File | Save All.
6. Провести рефакторинг.

Защита лабораторной работы заключается в предъявлении преподавателю полученных результатов (на экране монитора), демонстрации полученных навыков и ответах на вопросы преподавателя.

## **9. Вопросы.**

1. Какие этапы жизненного цикла разработки программного проекта поддерживает среда Visual Studio 2013.
2. Какие UML-диаграммы можно построить с использованием Visual Studio 2013.
3. Каковы возможности автоматической генерации кода в Visual Studio 2013.
4. Рефакторинг в Visual Studio 2013. Его возможности и ограничения.

## **ЛАБОРАТОРНАЯ РАБОТА №3.**

### **Выбор платформы и декомпозиция проекта. Основы использования системы контроля версий Git.**

**Цель работы:** получить навык выработки требований к программно-аппаратной платформе разрабатываемого ПО, проведения процесса декомпозиции задачи, освоить работу в системе контроля версий.

**Продолжительность работы – 4 часа.**

#### **Содержание**

1. Понятие "платформы" при разработке ПО .....	44
2. Декомпозиция .....	48
3. Система управления версиями .....	52
4. Порядок выполнения лабораторной работы .....	71
5. Вопросы .....	72

#### **1. Понятие платформы**

В информационных технологиях под термином «платформа» в широком смысле обычно понимается совокупность следующих компонентов:

- аппаратного решения;
- операционной системы (ОС);
- прикладных программных решений и средств для их разработки.

В более узком смысле выделяют следующие виды платформ:

**Программная платформа** – это совокупность операционной системы, средств разработки прикладных программных решений и прикладных программ, работающих под управлением этой операционной системы

**Прикладная платформа** – это средства выполнения и комплекс технологических решений, используемых в качестве основы для построения определенного круга прикладных программ

**Аппаратная платформа (hardware)** – это совокупность совместимых аппаратных решений с ориентированной на них операционной системой

Понятие «аппаратная платформа» связано с решением фирмы IBM о выработке и утверждении единого стандарта на основные комплектующие персонального компьютера. Стандарт получил название *платформа IBM PC-совместимых ПК*.

В настоящее время фирма IBM чаще используют понятие «платформа Wintel», подразумевая под этим сочетание микропроцессора фирмы Intel с операционной системой Windows. Микропроцессор при этом рассматривается как основа аппаратной платформы, которая определяет архитектуру персонального компьютера, т. е. его тип и характеристики.

Однако термин Wintel не совсем точно определяет понятие платформы, так как открытая архитектура современных IBM-совместимых персональных компьютеров позволяет собирать их из комплектующих, изготавливаемых различными фирмами-производителями, включая и микропроцессоры, которые в настоящее время выпускаются не только фирмой Intel, но и Advanced Micro Devices (AMD), Cyrix Corp.

и др. Кроме того, IBM-совместимые ПК могут работать не только под управлением операционной системы Windows, но и под управлением других операционных систем.

Платформа IBM-совместимых компьютеров включает в себя широкий спектр самых различных персональных компьютеров: от простейших домашних до сложных серверов.

Кроме платформы IBM-совместимых ПК достаточно широкое распространение получила *платформа Apple*, представленная довольно популярными на Западе компьютерами Macintosh.

Специалисты по компьютерной истории отдают приоритет в создании ПК именно компании Apple. С середины 70-х г. эта фирма представила несколько десятков моделей ПК — начиная с Apple I и заканчивая современным iMac, — и уверенно противостоит корпорации IBM.

Существует два основных варианта решения проблемы совместимости компьютерных платформ:

1. *Аппаратные решения* — это специальные платы, несущие на себе дополнительные процессор, оперативную память и видеопамять другой аппаратной платформы. Фактически они представляют собой отдельный компьютер, вставленный в существующий ПК. Его, как и обычный компьютер, можно оснастить любой операционной системой по выбору пользователя и соответствующим программным обеспечением. При этом можно легко переключаться между двумя операционными системами, обмениваться между ними файлами и выполнять другие операции, причем производительность обеих систем остается высокой и они не влияют друг на друга, так как практически не имеют разделяемых ресурсов, кроме мыши, клавиатуры и монитора. Основным недостатком таких плат является их высокая стоимость, хотя и несколько меньшая, чем отдельного ПК.

2. *Программные решения* — это специально написанные программы-эмулаторы, позволяющие запустить программное обеспечение, разработанное для персональных компьютеров одного типа, на другом ПК.

### **Прикладные решения и средства их разработки**

Средства разработки прикладных решений — это очень важная часть платформы персонального компьютера. От гибкости, богатства, удобства и надежности этих средств зависит популярность платформы. Платформа без средств разработки приложений под неё перестает существовать.

Все поставщики платформ поставляют и средства разработки прикладных решений в той или иной форме. Производители операционных систем предлагают всевозможные компиляторы и интерпретаторы, системы управления базами данных, системы организации взаимодействия (например, электронная почта). Конечно, решения для популярных операционных систем предлагают не только фирмы-создатели, но и другие фирмы-разработчики.

Для платформ, у которых возможности осуществления разработки решений непосредственно на них ограничены (например, для сотовых телефонов), производители предлагают средства разработки, функционирующие под популярной и мощной операционной системой (Windows, Linux). В дополнение к этим средствам предлагается эмулятор целевой платформы, на котором можно отладить решение, не используя целевую платформу непосредственно.

В настоящее время набирают популярность решения, обеспечивающие независимость разрабатываемых прикладных решений не только от аппаратной составляющей платформы, но и от операционной системы. Самые популярные решения подобного рода — Java и .Net.

Основная идея этих платформ состоит в создании «виртуальной машины» — специального программного комплекса, функционирующего на конкретной аппаратной платформе и на конкретной операционной системе. Прикладную программу обрабатывает виртуальная машина, которая преобразует «виртуальные команды» в команды конкретной программно-аппаратной платформы. В итоге получается, что программа для виртуальной машины функционирует на множестве связок «аппаратная часть—операционная система» без переделки. Единственное условие — наличие виртуальной машины для конкретного программно-аппаратного решения. Самая распространенная аппаратно-независимая платформа — Java.

Существует определенный класс программных продуктов — конструкторов, использование которых ограничено какой-либо предметной областью. Эти продукты реализуют не только базовую функциональность, но и гибкие средства создания решений в определенной области деятельности. Такие программные продукты зачастую называются прикладными платформами.

Под прикладной платформой понимаются среда исполнения и набор технологических решений, используемых в качестве основы для построения определенного круга приложений. Фактически приложения базируются на нескольких платформах, образующих многослойную среду. При этом важно, что платформа предоставляет разработчику определенную модель, как правило, изолирующую его от понятий и подробностей более низкоуровневых технологий и платформ.

Ключевым качеством прикладной платформы является достаточность ее средств для решения задач, стоящих перед бизнес-приложениями. Это обеспечивает хорошую согласованность всех технологий и инструментов, которыми пользуется разработчик. Другой важный момент — стандартизация. Наличие единой прикладной платформы для большого количества прикладных решений способствует формированию общего «культурного слоя», включающего и людей (программистов, аналитиков, пользователей), и методологию (типовые структуры данных, алгоритмы, пользовательские интерфейсы). Опираясь на этот «культурный слой», разработчик тратит минимум усилий на поиск необходимого решения практически в любой ситуации, начиная от включения в проект нового специалиста и кончая реализацией какой-либо подсистемы бизнес-приложения по типовой методологии.

Типичный представитель специальных прикладных платформ — система «1С: Предприятие». Сама по себе система является гибким, настраиваемым под нужды конкретного предприятия конструктором, предоставляющим разработчику решения, «более прикладные» методы и средства по сравнению с традиционными языками программирования, т. е. такая платформа представляет собой набор различных механизмов, используемых для автоматизации экономической деятельности и не зависящих от конкретного законодательства и методологии учета.

Существуют комплексные прикладные системы масштаба корпораций, которые являются основой для надежного ведения крупного бизнеса, так называемые ERP-системы (Enterprise Resource Planning Systems). Эти системы также являются прикладной платформой, гибко настраиваемой в своей предметной области.

### **Критерии выбора платформы**

Выбор платформы представляет собой сложную задачу, которая состоит из двух частей:

1. Определение сервиса, который должен обеспечиваться платформой
2. Определение уровня сервиса, который может обеспечить данная платформа

Существует несколько причин, в силу которых достаточно сложно оценить возможности платформы с выбранным набором компонентов, которые включаются в систему:

- подобная оценка прогнозирует будущее: предполагаемую комбинацию устройств, будущее использование программного обеспечения, будущих пользователей;
- конфигурация аппаратных и программных средств связана с определением множества разнородных по своей сути компонентов системы, в результате чего сложность быстро увеличивается;
- скорость технологических усовершенствований аппаратных средств, функциональной организации системы, операционных систем очень высокая и постоянно растет. Ко времени, когда какой-либо компонент широко используется и хорошо изучен, он часто рассматривается как устаревший.
- доступная потребителю информация об аппаратном обеспечении, операционных системах, программном обеспечении носит общий характер. Структура аппаратных средств, на базе которых работают программные системы, стала настолько сложной, что эксперты в одной области редко являются таковыми в другой.

Выбор той или иной платформы и конфигурации определяется рядом критериев. К ним относятся:

1. Отношение стоимость-производительность.
2. Надежность и отказоустойчивость.
3. Масштабируемость.
4. Совместимость и мобильность программного обеспечения.

*Отношение стоимость-производительность.* Появление любого нового направления в вычислительной технике определяется требованиями компьютерного рынка. Поэтому у разработчиков компьютеров нет одной единственной цели.

*Майнфрейм — это электронно-вычислительная машина, относящаяся к классу больших ЭВМ с высокой производительностью, поддерживающая многопользовательский режим работы для решения специализированных задач.*

Майнфрейм или суперкомпьютер стоят дорого, т.к. для достижения поставленных целей при проектировании высокопроизводительных конструкций приходится игнорировать стоимостные характеристики.

Другим крайним примером может служить низкостоимостная конструкция, где производительность принесена в жертву для достижения низкой стоимости. К этому направлению относятся персональные компьютеры. Между этими двумя крайними направлениями находятся конструкции, основанные на отношении стоимость-производительность, в которых разработчики находят баланс между стоимостными параметрами и производительностью. Типичными примерами такого рода компьютеров являются мини-компьютеры и рабочие станции.

*Надежность и отказоустойчивость.* Важнейшей характеристикой аппаратной платформы является надежность. Повышение надежности основано на принципе предотвращения неисправностей путем снижения интенсивности отказов и сбоев за счет применения электронных схем и компонентов с высокой и сверхвысокой степенью интеграции, снижения уровня помех, облегченных режимов работы схем, обеспечение тепловых режимов их работы, а также за счет совершенствования методов сборки аппаратной части персонального компьютера.

Введение отказоустойчивости требует избыточного аппаратного и программного обеспечения. Структура многопроцессорных и многомашинных систем приспособлена к автоматической реконфигурации и обеспечивает возможность продолжения работы системы после возникновения неисправностей. Понятие надежности включает не только аппаратные средства, но и программное обеспечение. Главной целью повышения надежности систем является целостность хранимых в них данных.

**Отказоустойчивость** – это свойство вычислительной системы, которое обеспечивает возможность продолжения действий, заданных программой, после возникновения неисправностей.

**Масштабируемость** должна обеспечиваться архитектурой и конструкцией компьютера, а также соответствующими средствами программного обеспечения.

Добавление каждого нового процессора в действительно масштабируемой системе должно давать прогнозируемое увеличение производительности и пропускной способности при приемлемых затратах. В действительности реальное увеличение производительности трудно оценить заранее, поскольку оно в значительной степени зависит от динамики поведения прикладных задач.

Возможность масштабирования системы определяется не только архитектурой аппаратных средств, но и зависит от заложенных свойств программного обеспечения. Простой переход, например, на более мощный процессор может привести к перегрузке других компонентов системы. Это означает, что действительно масштабируемая система должна быть сбалансирована по всем параметрам.

*Совместимость и мобильность программного обеспечения.* В настоящее время одним из наиболее важных факторов, определяющих современные тенденции в развитии информационных технологий, является ориентация компаний-поставщиков компьютерного оборудования на рынок прикладных программных средств. Это объясняется прежде всего тем, что для конечного пользователя в конце концов важно программное обеспечение, позволяющее решить его задачи, а не выбор той или иной аппаратной платформы. Переход от однородных сетей программно-совместимых компьютеров к построению неоднородных сетей, включающих компьютеры разных фирм-производителей, в корне изменил и точку зрения на саму сеть: из сравнительно простого средства обмена информацией она превратилась в средство интеграции отдельных ресурсов — мощную распределенную вычислительную систему, каждый элемент которой лучше всего соответствует требованиям конкретной прикладной задачи.

Этот переход выдвинул ряд новых требований:

Во-первых, такая вычислительная среда должна позволять гибко менять количество и состав аппаратных средств и программного обеспечения в соответствии с меняющимися требованиями решаемых задач.

Во-вторых, она должна обеспечивать возможность запуска одних и тех же программных систем на различных аппаратных платформах, т. е. обеспечивать мобильность программного обеспечения.

В-третьих, эта среда должна гарантировать возможность применения одних и тех же человеко-машинных интерфейсов на всех компьютерах, входящих в неоднородную сеть.

## 2. Декомпозиция

Выделение классов и объектов – одна из самых сложных задач объектно-ориентированного проектирования, которая осуществляется в процессе декомпозиции ключевых абстракций программной системы.

Декомпозиция занимает центральное место в объектно-ориентированном анализе и проектировании программного обеспечения. Под объектно-ориентированной декомпозицией понимается процесс разбиения системы на части, соответствующие объектам предметной области. Правильное разделение системы на составные части, представление предметной области в виде набора объектов, или разбиение программы на модули является почти такой же сложной задачей, как выбор правильного набора абстракций.

Разработчики программного обеспечения активно пользуются следующим правилом при выделении модулей программ: «Особенности системы, подверженные изменениям, следует скрывать в отдельных модулях; в качестве межмодульных можно использовать только те элементы, вероятность изменения которых мала». Также необходимо отметить, что поскольку модули служат элементарными и неделимыми блоками программы, которые могут использоваться повторно, это должно учитываться при распределении классов и объектов по модулям.

С проблемой декомпозиции исследователи ранее сталкивались при проектировании и анализе сложных административных, хозяйственных, производственных систем. В этих процессах очень часто непосредственное изучение объекта в целом как системы невозможно из-за его сложности. В этих случаях приходится расчленять объект на конечное число частей, учитывая связи между ними, характеризующие их взаимодействие. Здесь и начинается интерпретация исследуемого объекта как сложной системы, а его частей – как подсистем.

Если некоторые подсистемы оказываются все еще чрезмерно сложными, каждая из них разделяется (с сохранением связей) на конечное число более мелких подсистем. Процедура разделения подсистем продолжается до получения таких подсистем, которые в условиях данной задачи будут признаны достаточно простыми и удобными для непосредственного изучения. Эти подсистемы, не подлежащие дальнейшему расчленению, назовем элементами сложной системы.

Таким образом, в общем случае сложная система представляется как многоуровневая конструкция из взаимодействующих элементов, объединяемых в подсистемы различных уровней. Разделение системы на элементы в общем случае может быть выполнено неоднозначным образом и является в высшей степени условным.

Естественно возникает вопрос о правилах разделения сложных систем или декомпозиции в случае разработки объектно-ориентированной модели для задач, поставленных для заданной предметной области.

### **Синтез и анализ в изучении сложных систем**

Разделение или декомпозиции применяется в случаях, когда рассматривается существующая система или проектируется новая. При этом применяются два фундаментальных понятия системологии: анализ (как было сформулировано – рассмотрение) и синтез (проектирование) систем. Задачи анализа определяются как изучение свойств и поведения системы в зависимости от ее структуры и значений параметров, исходя из заданных свойств системы; задачи синтеза сводятся к выбору структуры и значений параметров, исходя из заданных свойств системы.

В настоящее время не существуют формальных методов синтеза сложных систем. По-видимому, это в принципе невозможно, так как сложная система синтезируется для условий, которые невозможно описать конкретными математическими моделями в силу непрерывно изменяющегося многообразия условий. Поэтому повсеместно на практике применяется синтез через анализ, когда задается исходя из практических соображений некоторая структура системы, затем она анализируется, затем изменяются ее параметры или модифицируется структура, затем осуществляется анализ и так далее до достижения необходимого результата.

Другими словами, при разработке достаточно сложного программного обеспечения решается задача синтеза методами синтеза через анализ. С одной стороны, задача создания новой структуры или задача изобретения новых абстракций является

творческой, с другой стороны, существуют общие правила, помогающие в этом процессе.

### **Обобщенное правило декомпозиции**

Зададим в общем виде функционал эффективности системы в виде следующего выражения:  $A = F(a_1, \dots, a_n)$ , где  $A$  – показатель, отражающий качество реализации основного назначения системы (например, некоего системного качества или эмергентного свойства),  $a_1, \dots, a_n$  – параметры, от которых зависит данное системное качество (они также могут быть функционально зависимыми от других параметров).

В качестве пояснения назначения функционала рассмотрим задачу увеличения высоты полета самолета. Пусть  $A$  – высота полета самолета. Первый вопрос, на который должен ответить человек, решающий данную задачу – от чего зависит высота полета? Для простоты рассмотрим два составляющих элемента самолета: двигатель, крылья, и, кроме того, саму систему – самолет. Выделим свойства, оказывающие влияние на заданное системное свойство, например, в двигателе – мощность (пусть это  $a$ ), в крыльях – площадь ( $c$ ), а в самолете его вес ( $b$ ).

Значит, для увеличения системного качества  $A$  нужно рассмотреть возможности синтеза такой структуры рассмотренных элементов при которой значения показателей  $a, b, c$  приведут к требуемому результату. Это очевидный пример синтеза через анализ.

Конечно же, для увеличения высоты полета самолета необходимо рассматривать более сложный комплекс показателей. И более того, нельзя забывать о возможной функциональной зависимости данных параметров и о влиянии других (неучтенных) параметров. Вполне может оказаться так, что один из неучтенных параметров оказывает большее влияние на анализируемое системное качество  $A$ .

Таким образом, в общем виде правило декомпозиции заключается в выполнении следующих основных этапов:

1. Определение системных(ого) свойств(а), значимых (ого) для решения поставленной задачи. В примере – высота полета самолета.
2. Поиск составных частей системы и их свойств, оказывающих решающее влияние на выделенные системные свойства. В примере – двигатель (мощность), крылья (площадь), самолет (вес).

Декомпозиция может продолжаться до тех пор, пока не будут выделены достаточно простые составляющие рассматриваемой системы и её подсистемы. Здесь «простота» – является субъективным показателем, то есть каждый аналитик определяет сам для себя как долго необходимо выполнять разбиение системы и ее подсистем на составляющие элементы. Выделенные элементы и их упрощенное описание в виде набора свойств, необходимых для решения поставленной задачи представляет собой не что иное как абстракцию.

### **Особенности проектирования программных систем**

Необходимо отметить, что сами по себе объекты не представляют интереса: только в процессе взаимодействия объектов реализуется система. Поведение системы определяется сочетанием состояний объектов, которые свою очередь задаются некоторым состоянием агрегируемых объектов. Поэтому на ранних стадиях проектирования внимание проектировщика сосредотачивается на внешних проявлениях системы, что позволяет создать ключевые абстракции и механизмы.

Такой подход создает логический каркас системы: структуру классов, представляющий объекты реальной системы в заданной абстракции. На последующих фазах проектирования (включая и реализацию – где происходит распределение классов

по модулям), внимание переключается на внутреннее поведение ключевых абстракций и механизмов, а также их физическое представление. Принимаемые в процессе проектирования решения задают архитектуру системы, архитектуру процессов и архитектуру модулей.

*Архитектура* – это логическая и физическая структура системы, сформированная всеми стратегическими и тактическими проектными решениями. Хорошей архитектуре присущие следующие основные черты:

- многоуровневая система абстракции;
- модульность;
- простота.

Многоуровневая система абстракции основывается на иерархии классов, максимально взаимосвязанных по горизонтали и минимально по вертикали. Каждый уровень абстракции «сотрудничает» по вертикали с другим посредством четкого интерфейса с внешним миром и основываются на столь же хорошо продуманных средствах нижнего уровня. На каждом уровне интерфейс абстракции строго ограничен от реализации. Реализацию можно изменять, не затрагивая при этом интерфейс. Изменяясь внутренне, абстракции продолжают соответствовать ожиданиям внешних клиентов.

Модульность – это свойство программной системы, которая была разделена на связные и слабо зацепленные между собой модули, реализующие заданные абстракции разрабатываемой системы. Модуль – это единица кода, служащая строительным блоком физической структуры системы; программный блок, который содержит объявления, выраженные в соответствие с требованиями языка и образующие физическую реализацию части или всех классов и объектов логического проекта системы. Как правило, модуль состоит из интерфейсной части и реализации.

Как уже отмечалось выше, связность – это степень взаимодействия между элементами отдельного модуля (а для объектно-ориентированной декомпозиции еще и отдельного класса или объекта). Наиболее желательной является функциональная связность, при которой все элементы класса или модуля тесно взаимодействуют в достижении определенной цели.

Для построения системы должен использоваться минимальный набор неизменяемых компонент; сами компоненты должны быть по возможности стандартизованы и рассматриваться в рамках единой модели. Применительно к объектно-ориентированному проектированию такими компонентами являются классы и объекты, отражающие ключевые абстракции системы, а единство обеспечивается соответствующими механизмами реализации.

Архитектура считается простой, если она не содержит ничего лишнего, а общее поведение системы достигается общими абстракциями и механизмами. Необходимо отметить, что не существует единственно – верного способа классификации абстракции и проектирования архитектуры.

## **Системологическое описание проектирования программных систем**

На основе изложенных особенностей проектирования можно говорить о проектирование как о процедуре синтеза программной системы на основе анализа реальной системы. По результатам проектирования программная система представляет собой взаимосвязанный комплекс подсистем, где каждая подсистема в терминах объектно-ориентированного проектирования является модулем, представляемым либо объектом, либо группой взаимосвязанных объектов.

Здесь каждый объект является экземпляром заданной абстракции. Структурой системы являются устойчивые взаимосвязи между подсистемами. Деление системы на подсистемы и объекты является условным и зависит от объема и сложности самой системы.

Программная система имеет системные качества, или эмергентные свойства, отражающие основное ее назначение или основные функции, отсутствующие у входящих в неё подсистем и являющиеся интегральным результатом согласованного функционирования подсистем. Подсистемы, входящие в программную систему, также обладают системными качествами, или эмергентными свойствами, являющимися основным функциональным назначением подсистем. Агрегация одной подсистемы в другую формирует иерархию подсистем. Сочетание состояний объединенных подсистем формирует новые эмергентные свойства каждого из объединений.

Системологический подход, который, несомненно, лежит в основе объектно-ориентированного анализа, позволяет установить наиболее общие закономерности проектирования программных систем, представляющих реальные объекты и/или системы физического мира. Целенаправленное использование выявленных закономерностей при исследовании одних систем позволяет повысить эффективность анализа и синтеза других систем.

Так, в любом творческом процессе разработки системы есть общие похожие закономерности. В начале работы разработчик обосновывает необходимость, цель и варианты решений, на основе анализа существующих материальных и/или знаковых систем. Анализ системы начинается с изучения ее внешних проявлений и выделения системных или эмергентных свойств. Данные свойства формируют ключевые подсистемы, на основе которых формируется структура системы, обычно представляемая в виде иерархии подсистем. Такой подход создает логический каркас системы, задаваемый множеством связей между подсистемами и их свойствами, которые в результате формируют общее поведение системы и её системные свойства.

Следующий этап – синтез внутренней реализации каждой из выделенных подсистем. Здесь важно помнить, что любая подсистема должна иметь четко определенное и неизменное в процессе эксплуатации множество элементов управления (в терминах объектно-ориентированного проектирования – интерфейс) её поведением. Такая реализация подсистем позволяет влиять на эмергентные свойства системы путем замены одних внутренних реализаций выбранных подсистем на другие.

### **3. Система управления версиями**

Система управления версиями (от англ. Version Control System, VCS или Revision Control System) — программное обеспечение для облегчения работы с изменяющейся информацией. Система управления версиями позволяет хранить несколько версий одного и того же документа, при необходимости возвращаться к более ранним версиям, определять, кто и когда сделал то или иное изменение, и многое другое.

Такие системы наиболее широко используются при разработке программного обеспечения для хранения исходных кодов разрабатываемой программы. Однако они могут с успехом применяться и в других областях, в которых ведётся работа с большим количеством непрерывно изменяющихся электронных документов. В частности, системы управления версиями применяются в САПР, обычно в составе систем управления данными об изделии (PDM). Управление версиями используется в инструментах конфигурационного управления (*Software Configuration Management Tools*). Систем контролия версий существует довольно много, наиболее распространенные : RCS, CVS, Subversion, Aegis, Monoton, Git, Bazaar, Arch, Perforce, Mercurial, TFS.

В настоящий момент две наиболее используемые системы контроля версий — Git и Subversion (SVN). Принципиальная разница: GIT распределенная, а SVN — нет. Другими словами, если есть несколько разработчиков работающих с репозиторием у каждого на локальной машине будет ПОЛНАЯ копия этого репозитория. Разумеется есть и где-то центральная машина, с которой можно клонировать репозиторий. Это напоминает SVN. Основной плюс в том, что если вдруг у вас нет доступа к интернету, сохраняется возможность работать с репозиторием. Потом только один раз сделать синхронизацию и все остальные разработчики получат полную историю.

GIT сохраняет метаданные изменений, а SVN целые файлы. Это экономит место и время. Система создания branches, versions и прочее в GIT и SVN отличаются значительно. В GIT проще переключаться с ветки на ветку, делать слияние между ними. Недостаток Git — приватные репозитории являются платными, а в SVN есть возможность бесплатно создавать приватные репозитории с ограничением по количеству участников. Тем не менее, на данный момент более популярен Git, а также его форки, такие как, например, GitLab. Поэтому в данной лабораторной работе будет рассмотрена система управления версиями Git.

### Установка.

Для установки необходимо скачать файл устанавливающий GIT систему на ваш компьютер. Ссылка для скачивания: <http://git-scm.com/downloads>

Итак, что же представляет из себя репозиторий? Это папка, хранящая весь ваш проект. Репозиторий есть локальный (на вашем компьютере) и удаленный (на сервере). Именно с удаленного сервера в дальнейшем берется вся информация для выпуска конечного продукта. Изменения в репозиторий разработчики добавляют с помощью системы ветвления. В качестве примера мы разработаем простейшую программу с использованием базовых возможностей Git.

В данной лабораторной работе будут приведены примеры работы со стандартным графическим интерфейсом git gtk, а также для каждой команды будут приведены аналоги для консоли. Зачем это нужно? Порой разработчикам, в частности веб-разработчикам, приходится вносить изменения в файлах на удаленном компьютере, поэтому для экономии ресурсов компьютера и более быстрой работы используется подключение через консоль. Вы увидите, что в ряде случаев использование консольной версии Git оказывается более удобным нежели его графическая оболочка. Поэтому под каждым скриншотом будет соответствующая консольная команда. Работать будем с английской версией поскольку большинство информации в сети именно на английском, в случае возникновения проблем решение найти будет проще.

Установите git. Создайте пустую папку проекта. Затем внутри этой папки нажмите правую кнопку мыши и выберете пункт Git GUI Here. Для открытия консоли — Git Bash Here (Рис. 1, 2).

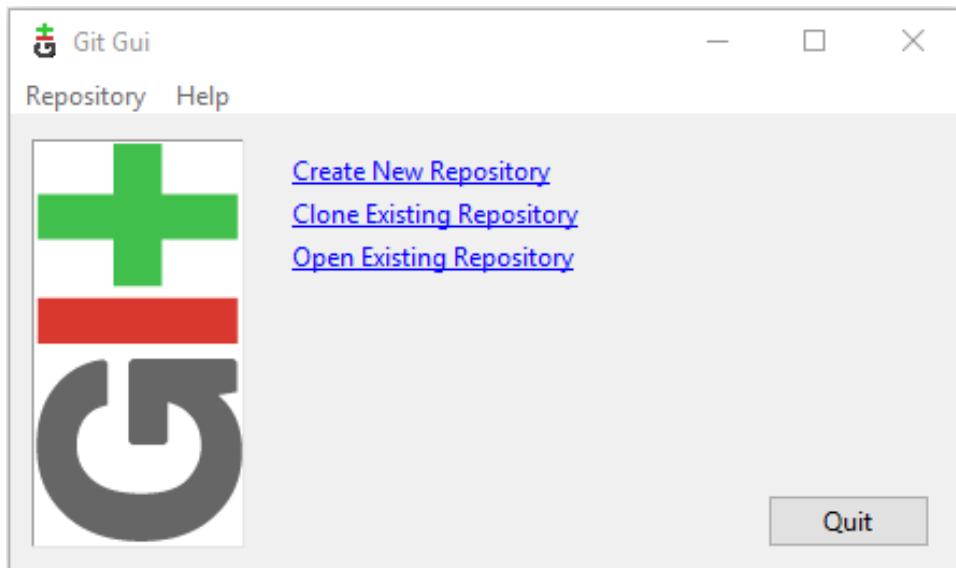


Рисунок 1. Создание пустой папки проекта

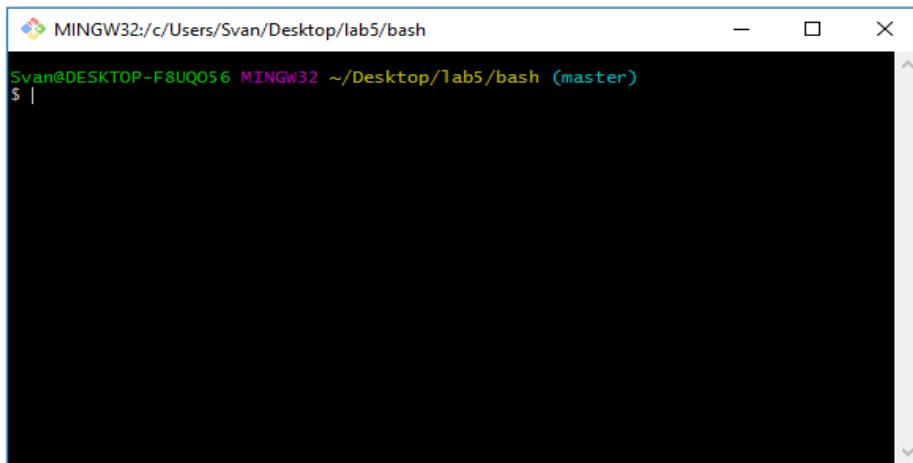


Рисунок 2. Создание пустой папки проекта на консоли

Нажмите Create New Repository, и через кнопку Browse выберите текущую папку.  
В ней создастся пустой репозиторий Git (Рис 3.).

Консоль: **git init**

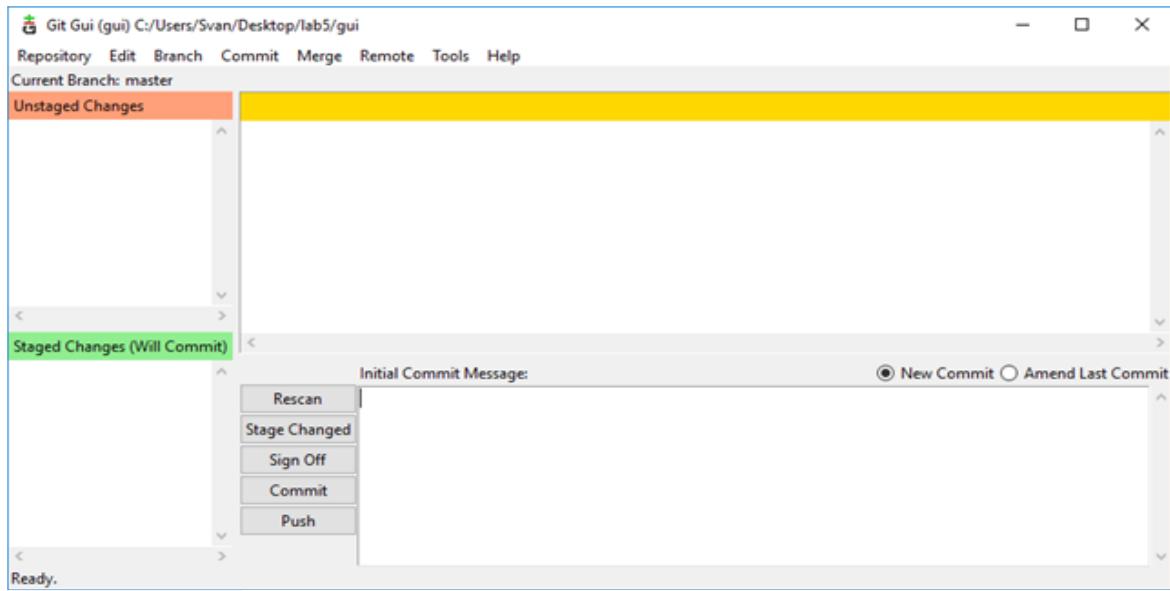


Рисунок 3. Создание пустого репозитория

В настоящий момент мы находимся на ветке master. Эта ветка по умолчанию создается в Git. Добавим подпись к нашему репозиторию чтобы остальные разработчики знали кто вы. Для этого выберем пункт Edit – Options. В нашем случае (Рис. 4) удобно будет сделать универсальную подпись для всех репозиториев, поэтому заполним соответствующие поля в правой колонке (User Name и Email Address).

На консоли:

```
git config -- global user.name "Student"
git config -- global user.email yourmail@yourmail.ru
```

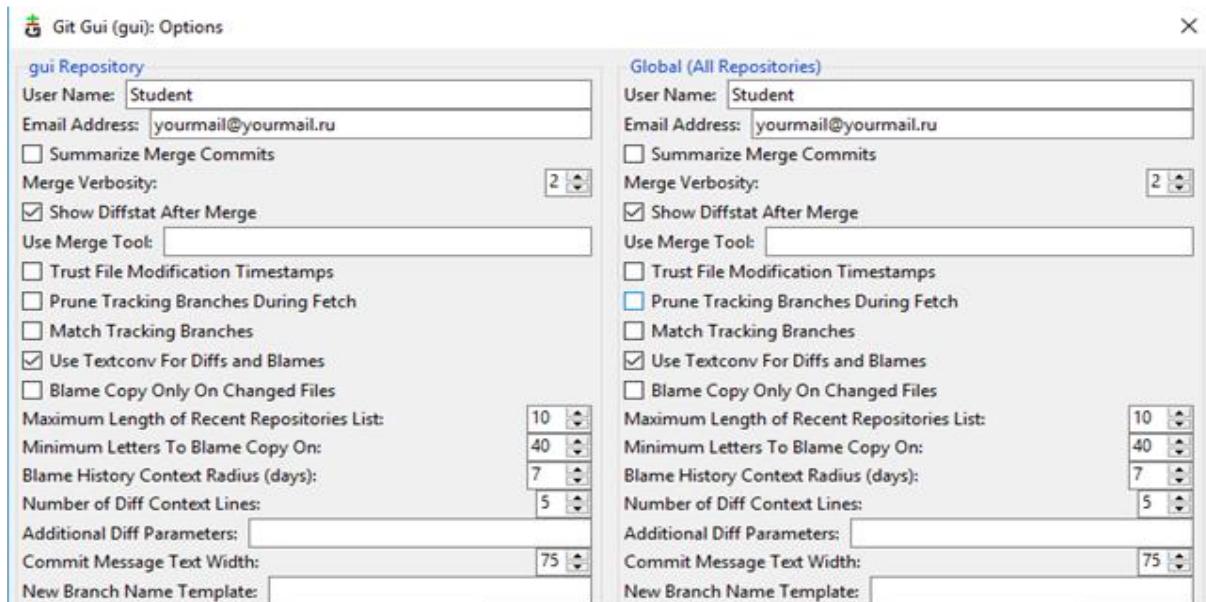


Рисунок 4. Создание подписей репозитория

Добавим в нашу папку файл main.cpp со следующим содержанием:

```
#include <iostream>
int main() {
    std::cout << "Hello world!\n";
    system("pause");
    return 0;
}
```

Нажмем кнопку Rescan и увидим появившийся файл в колонке Unstaged Changes. В этой колонке будут появляться все измененные файлы в репозитории (Рис. 5).

### Консоль: git status

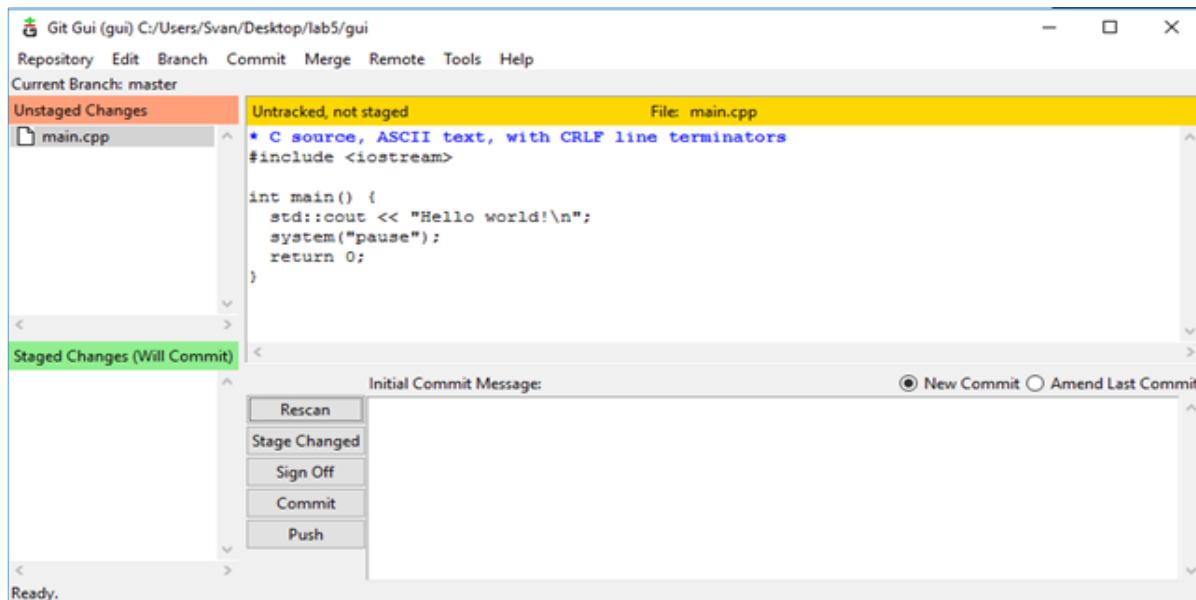


Рисунок 5. Добавление в папку файла main.cpp

Однако не всегда бывает надо запомнить изменения всех файлов. Чтобы Git добавил этот файл в список сохраняемых (в индекс) нужно нажать на иконку слева от названия файла. Файл переместится в зеленую зону Staged Changes.

На консоли:

```
git add (если нужно добавить все файлы в индекс).
git add filename (если нужно добавить конкретный файл).
```

Сохраним в репозитории наше первое изменение, иначе — сделаем коммит. Для этого в окне Initial Commit Message добавим описание коммита и нажмем кнопку Commit (Рис. 6). На консоли: **git commit -m "init"**

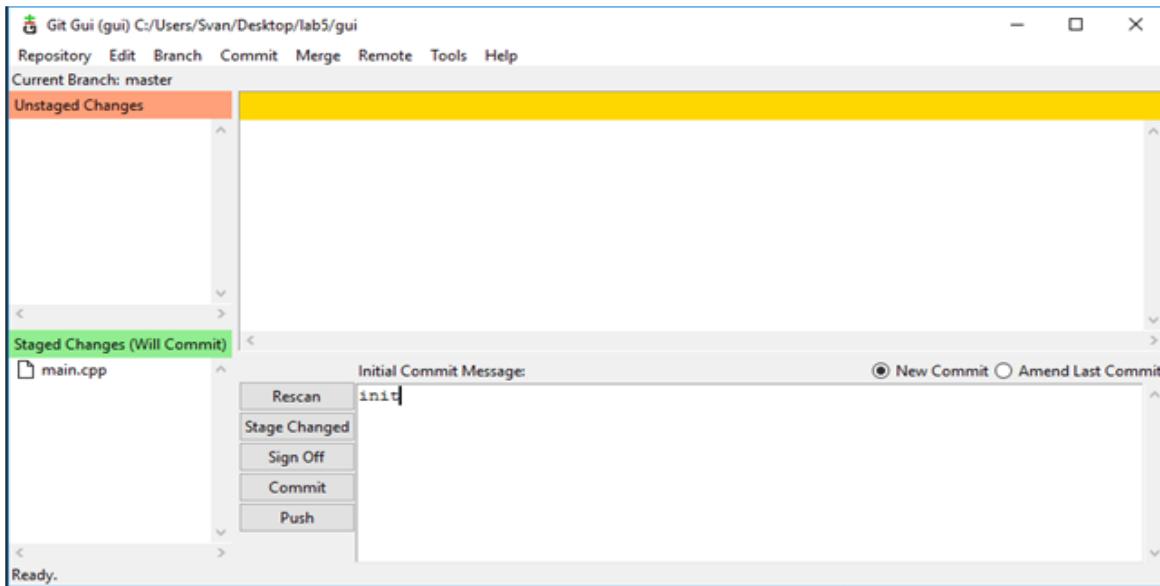


Рисунок 6. Первое изменение – коммит

Изменения сохранены. Но работать в ветке master небезопасно, поскольку там может оказаться неисправный код. Поэтому мы создадим ветку dev, в которой будет содержаться бета-версия нашей программы, и ветку student, в которой мы и будем работать. На практике разработчики используют для работы свои ветки, потом делают слияние в ветку dev, продукт из ветки dev тестируется и только после этого ветка dev вливается в master. Далее мы это рассмотрим. А пока создаем ветку dev. Для этого выбираем Branch – Create. (консоль: **git checkout -b dev**). Аналогично создадим ветку student (консоль: **git checkout -b student**) (Рис. 7).

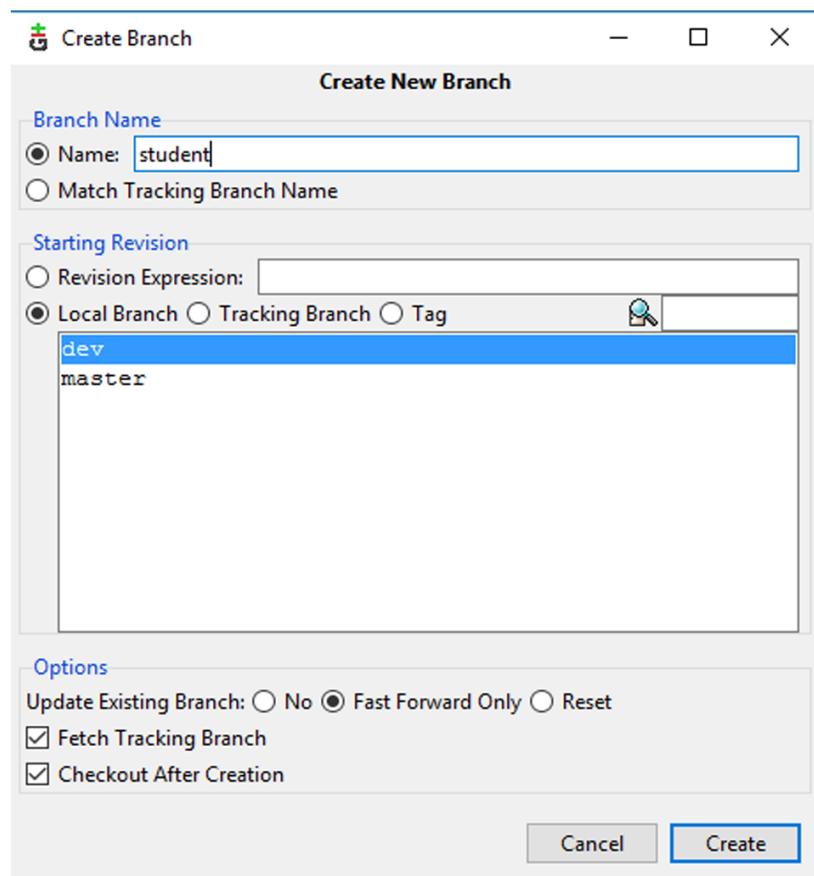
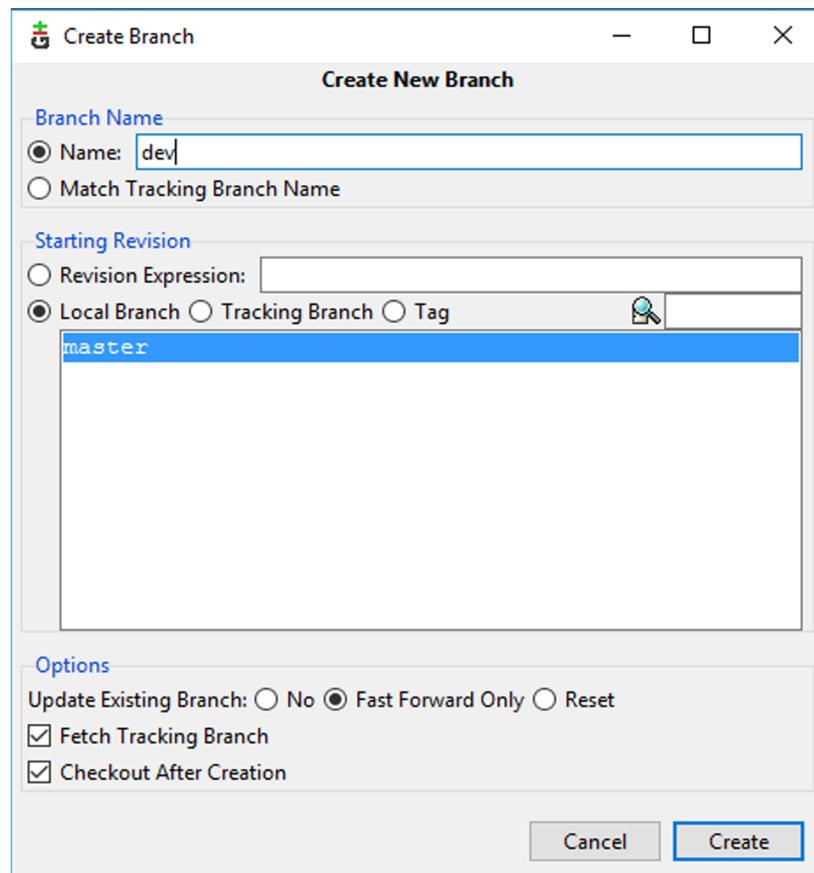


Рисунок 7. Создание рабочих веток dev и student

Теперь, находясь в рабочей ветке, мы можем спокойно вносить изменения в нашу программу (Рис. 8). Изменим main.cpp:

```
#include <iostream>
using namespace std;
int main() {
    char name[20];
    cout << "Enter your name: ";
    cin >> name;
    cout << "Hello, " << name << "!\n";
    system("pause");
    return 0;
}
```

Нажимаем Rescan  
(Консоль: **git status**)

И видим:

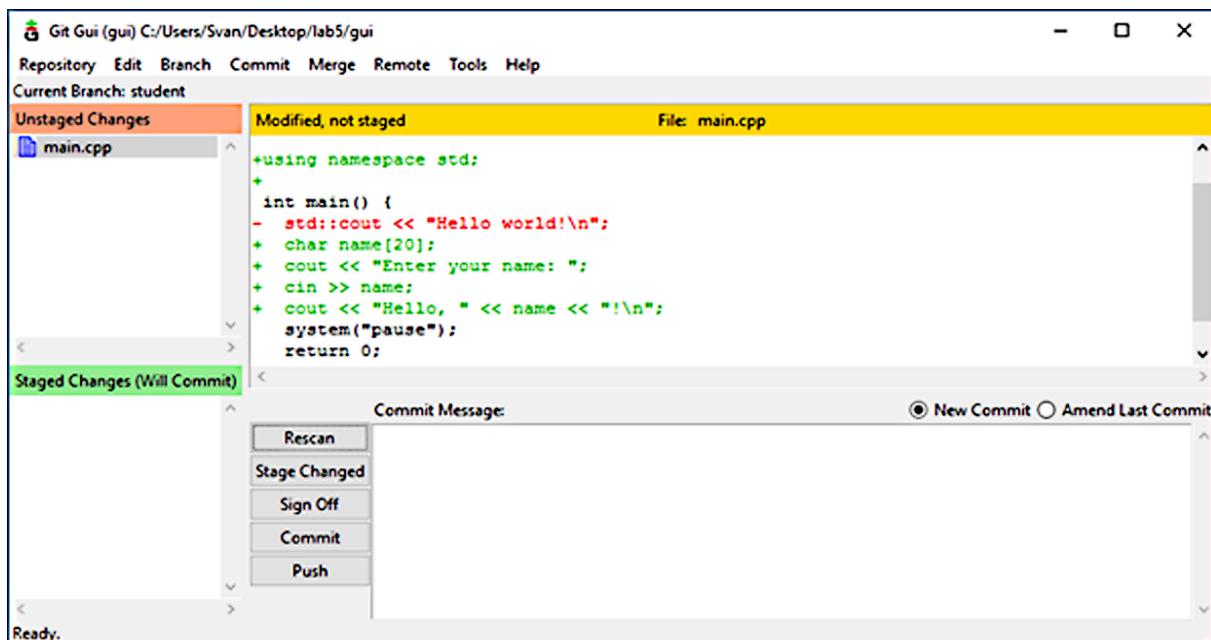


Рисунок 8. Внесение изменений в текст программы

Система заметила наши изменения и отобразила их красным цветом. Добавим файлы в индекс и сделаем коммит (Рис. 9).

Консоль:

```
git add .
git commit -m "added student name input"
```

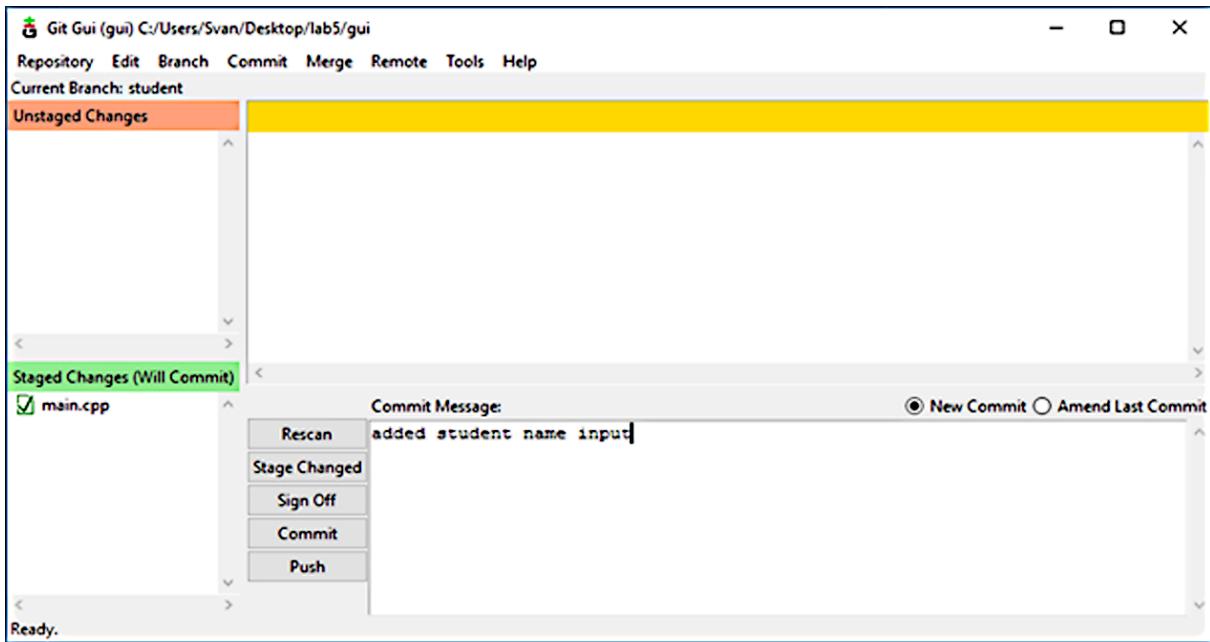


Рисунок 9. Делаем коммит

Внесем еще изменения в main.cpp:

```
#include <iostream>
using namespace std;
int main() {
    char name[20];
    cout << "Enter your name: ";
    cin >> name;
    cout << "Hello, " << name << "!\n";
    cout << "Nice to see you learning Git\n";
    system("pause");
    return 0;
}
```

Аналогично сделаем коммит с описанием "added greeting Git info"

Консоль:

```
git add .
git commit -m "added greeting Git info"
```

Представим, что тестовую ветку dev сейчас смотрит клиент и хочет чтобы кроме приветствия миру выводилось еще и пожелание хорошего дня. Что ж, желание клиента – закон, переходим на ветку dev и меняем содержимое файла main.cpp. Для этого выбираем Branch – Checkout и выбираем ветку dev (Рисунок 10).

Консоль:

```
git add .
git commit -m "added having a nice day wish"
```

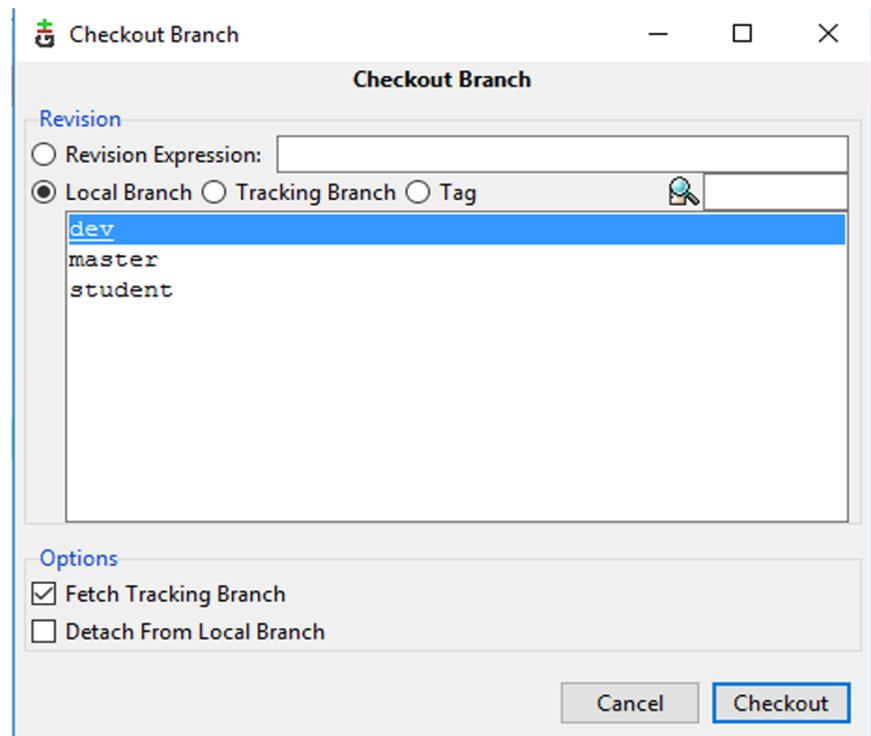


Рисунок 10. Выбор ветки dev для внесения изменений

Git предоставляет возможность посмотреть всю историю коммитов, иначе – лог. Для этого выбираем Repository – Visualize All Branch History (Рис. 11).

Консоль: `git log --graph --all`

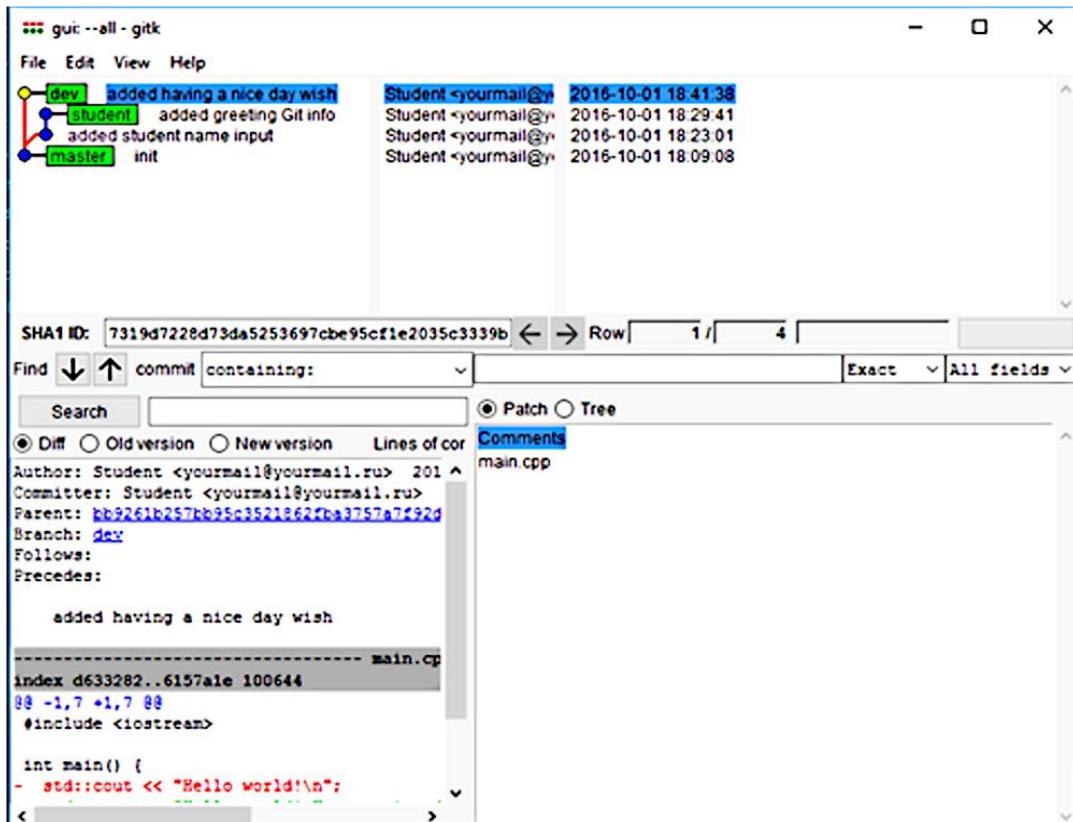


Рисунок 11. История коммитов

Попробуем теперь влить изменения с ветки student в ветку dev, другими словами – сделать merge в dev из student. Выбираем Merge – Local Merge (Рис.12)

Консоль: `git merge student`

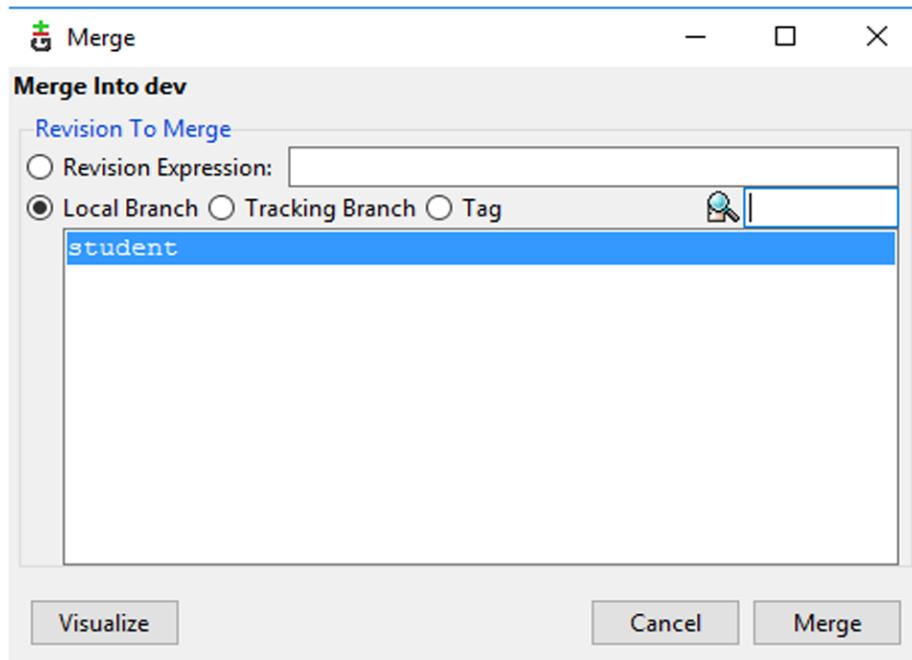


Рисунок 12. Делаем merge в dev из student

Но при попытке выполнить слияние возник конфликт (Рис. 13):

```
Git Gui (gui): Merge
merge student
Auto-merging main.cpp
CONFLICT (content): Merge conflict in main.cpp
Automatic merge failed; fix conflicts and then commit the result.
warning: old-style 'git merge <msg> HEAD <commit>' is deprecated.

Error: Command Failed
```

Рисунок 13. При попытке выполнить слияние возник конфликт

Тут мы дошли до самой частой ошибки разработчиков. Дело в том, что мы изменили файл main.cpp сначала в ветке student, а потом в ветке dev никак не синхронизировав изменения между собой. Поэтому Git не знает какие изменения

оставлять, а какие удалять. Этот конфликт нужно решить. Для этого откроем main.cpp и заменим его содержимое следующим:

```
#include <iostream>
using namespace std;
int main() {
    char name[20];
    std::cout << "Hello world! Have a nice day!\n";
    cout << "Enter your name: ";
    cin >> name;
    cout << "Hello, " << name << "!\n";
    cout << "Nice to see you learning Git\n";
    system("pause");
    return 0;
}
```

Мы решили конфликт, добавив вручную изменения из ветки student в наш файл, находящийся на ветке dev. Теперь нужно сделать коммит, дающий понять Git'у, что конфликт решен (Рис. 14).

Консоль: **git commit -m "conflict resolved"**

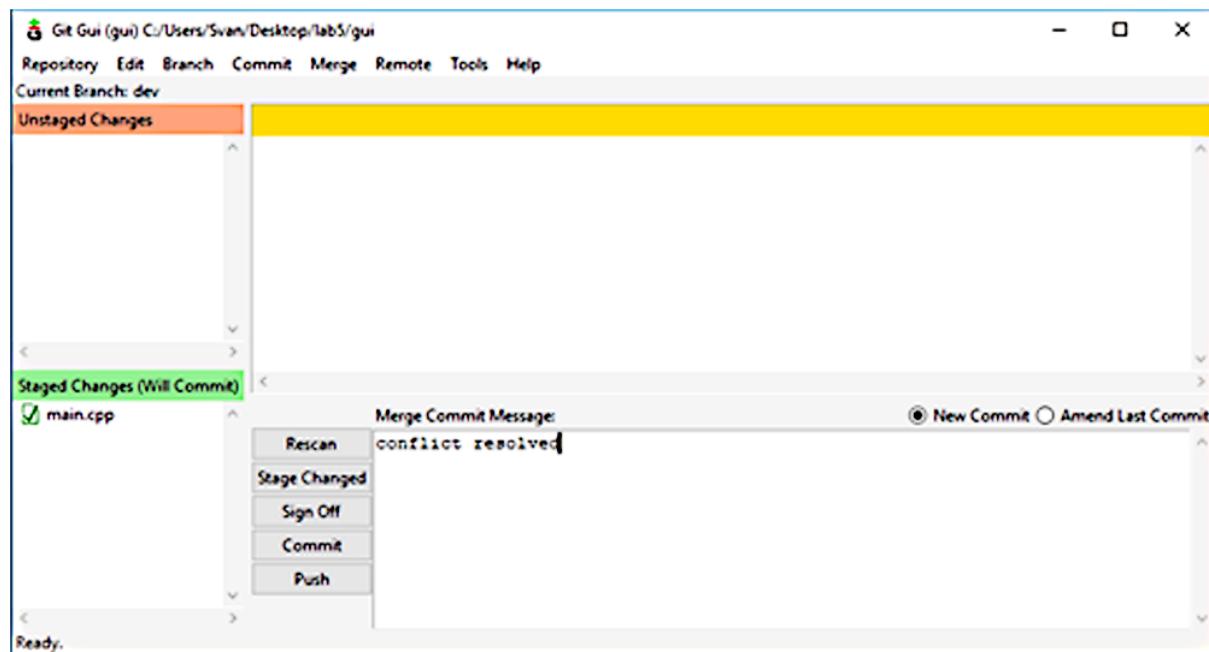


Рисунок 14. Делаем коммит, дающий понять Git'у, что конфликт решен

После этого делаем повторное слияние веток dev и student. На этот раз все проходит успешно. Можем проследить это на нашем дереве коммитов (Рис. 15).

Консоль: **git log --graph --all**

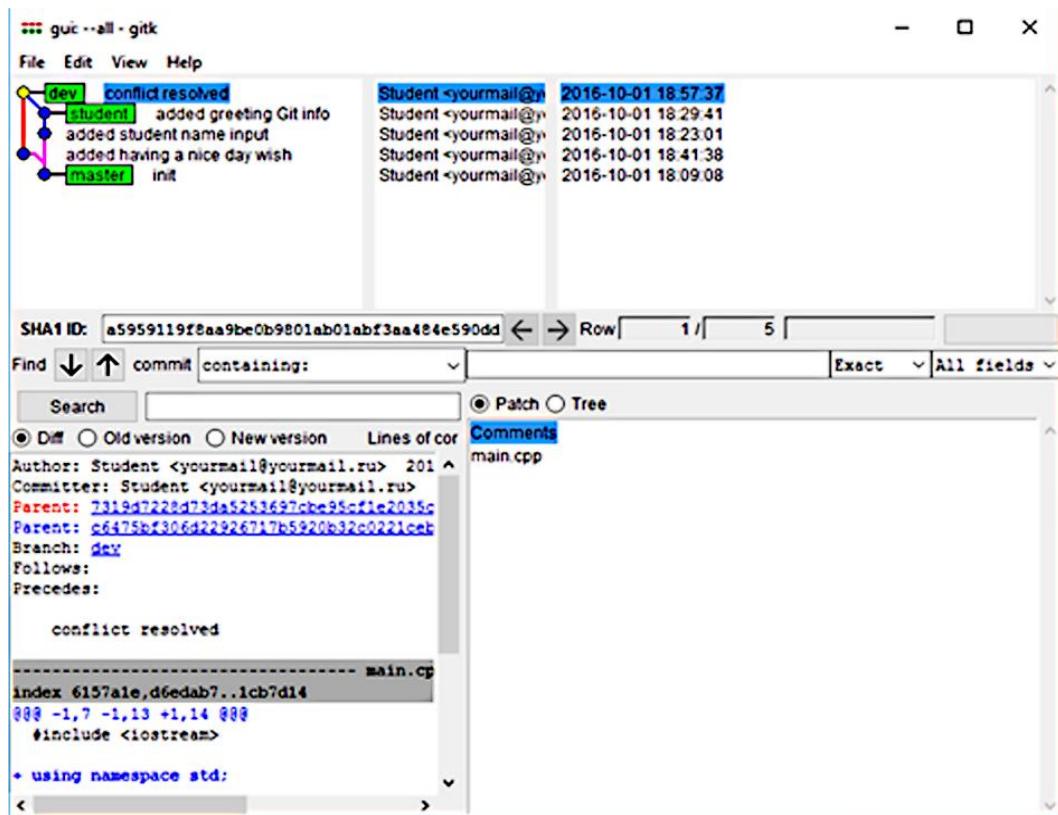


Рисунок 15. Повторное слияние веток dev и student

Проблем не замечено, можем сливать ветку dev в ветку master. Для этого переходим в ветку master через Branch – Checkout и после этого Merge – Local Merge

Консоль:

**git checkout master**

**git merge dev**

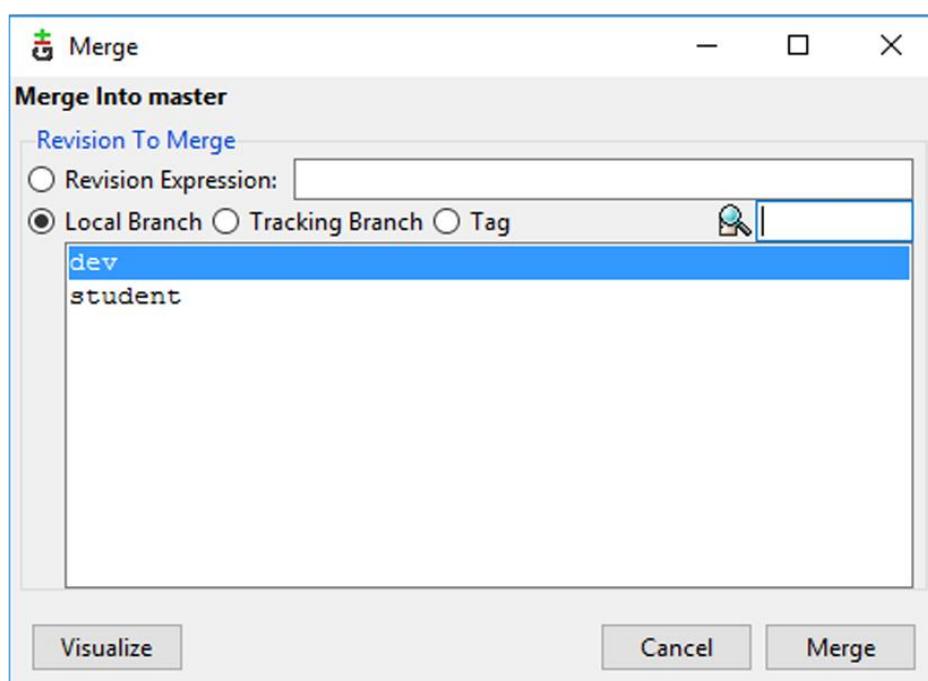


Рисунок 16. Ветка dev – в ветку master

Половина работы проделана. Теперь нужно вылить весь наш локальный репозиторий на удаленный. Для этого нужно перейти на [github.com](https://github.com), создать или войти в свою учетную запись. После этого создать репозиторий, выбрав соответствующий пункт в верхнем меню (Рис. 17).



Рисунок 17. Создать удаленного репозитория

Создав новый репозиторий, вы попадете на стартовую страницу, где есть интересующая нас ссылка HTTPS (Рис. 18).

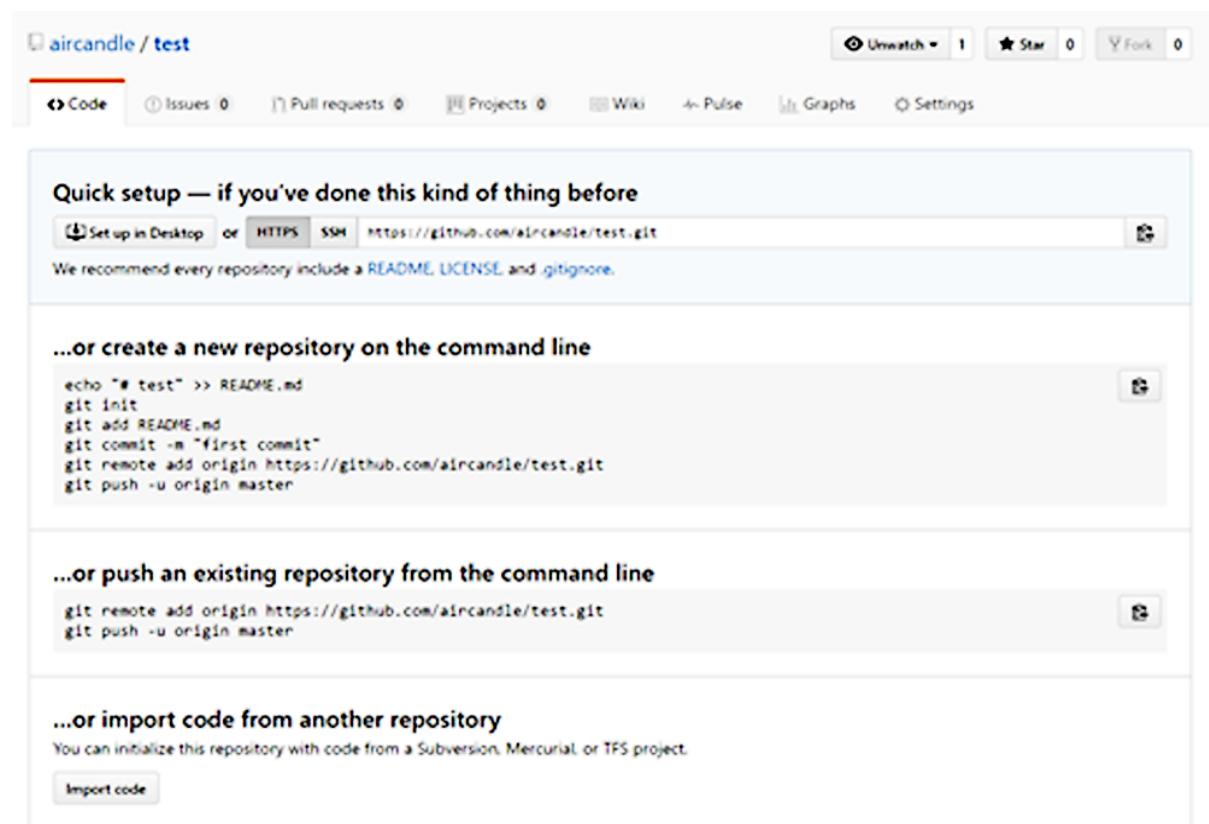


Рисунок 18. Стартовая страница со ссылкой HTTPS

Копируем эту ссылку и возвращаемся в Git GUI. Выбираем пункт Remote – Push, выделяем все наши ветки, вставляем в строку Arbitrary Location ссылку к нашему удаленному репозиторию и жмем кнопку Push (Рис. 19).

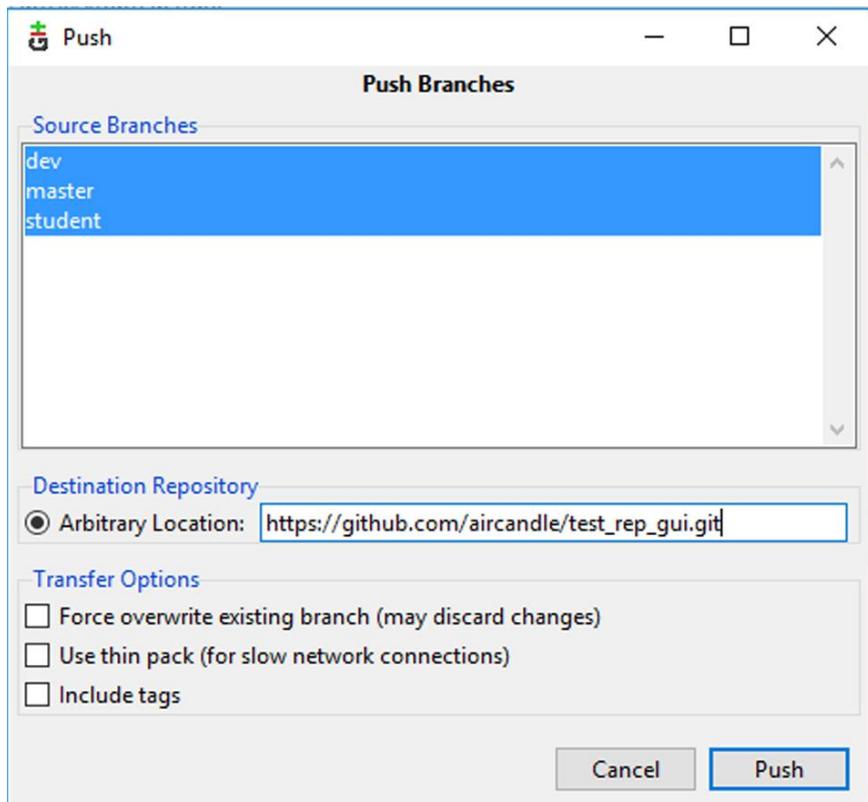


Рисунок 19. Вставляем ссылку на удаленный репозиторий

На практике удаленный репозиторий обычно добавляется в список, но мы это сделаем позже. А пока дожидаемся сообщения об успешной отправке нашего локального репозитория (Рис. 20).

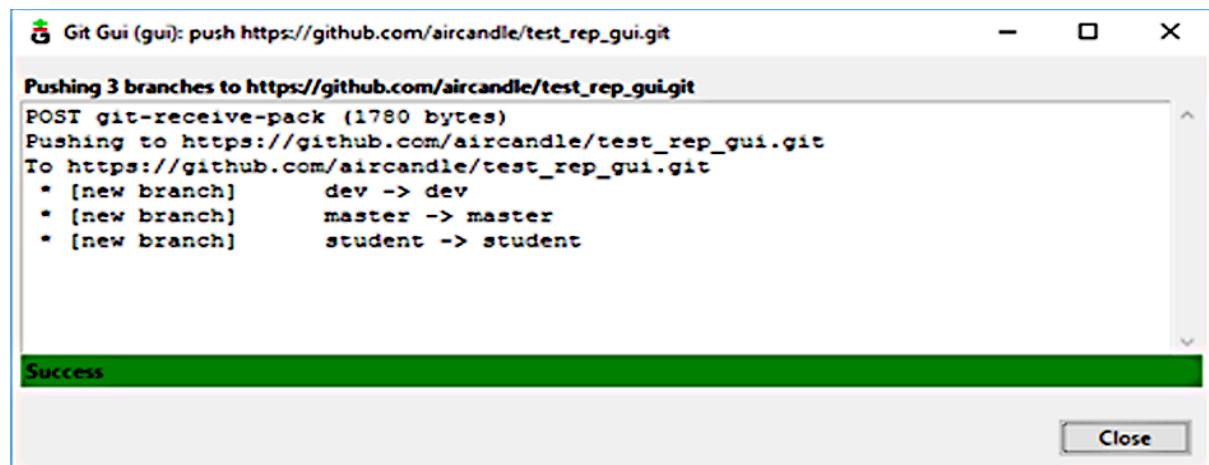


Рисунок 20. Отправка на удаленный репозиторий

Для проверки можем перейти в браузере на `github` и обновить страницу с репозиторием. Мы увидим, что все наши ветки и коммиты успешно перенесены.

Но обычно мы далеко не всегда первыми создаем репозитории, намного чаще возникает необходимость подключиться к уже существующему. Поэтому сейчас мы рассмотрим случай, когда у нас имеется только удаленный репозиторий и нам необходимо внести в нем правки. Для этого перейдем в директорию, в которой будет расположена папка с содержимым нашего проекта. Важный момент: если вы работаете

именно с Git GUI, то вы НЕ ДОЛЖНЫ создавать папку для вашего проекта. Git GUI создает ее самостоятельно, это особенность данного графического клиента. Если же вы предпочтете работать с консолью, то все понятнее и проще: создаете папку с проектом, там щелчок правой кнопкой мыши – Git Bash Here и спокойно выполняете все необходимые команды.

В нашем случае возникла необходимость изменить файл main.cpp, поскольку мы указали пространство имен и std::cout нужно заменить на cout.

Итак, в нужной папке кликаем правой кнопкой мышки – Git GUI Here. Выбираем пункт Clone Existing Repository, открывается диалоговое окно (Рис. 21).

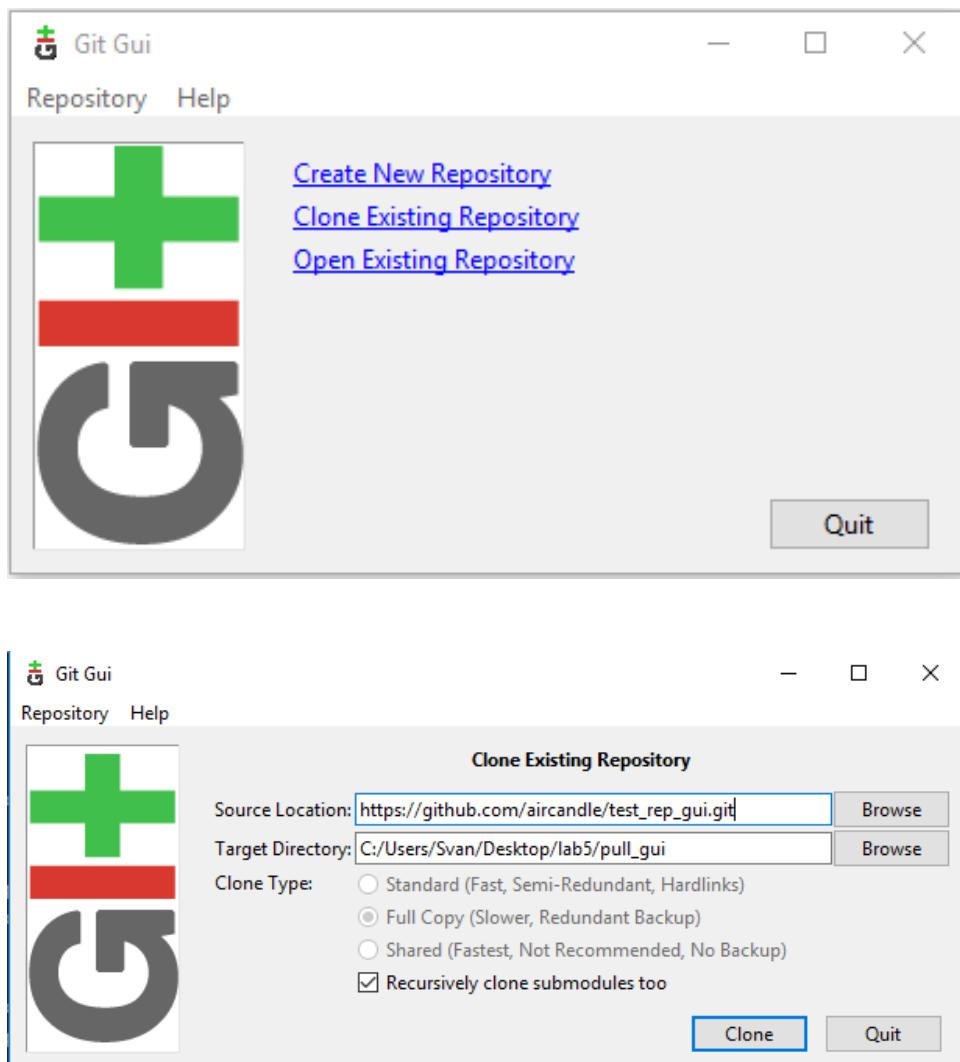


Рисунок 21. Размещение на удаленный репозиторий

В строку Source Location вставьте ссылку на удаленный репозиторий, в поле Target Directory через кнопку Browse выберите текущую папку. Как только путь к папке появится в строке вручную добавьте путь к новой НЕСУЩЕСТВУЮЩЕЙ папке с вашей программой. Например, здесь Git GUI вызывался из папки lab5, вручную необходимо было дописать /pull\_gui. После этого нажимаем кнопку Clone. Git подгрузит все данные из удаленного репозитория, создаст локальный репозиторий и синхронизирует локальный с удаленным.

Консольный аналог:

```
git init  
git remote add origin https://github.com/aircandle/test\_rep\_gui.git  
git pull
```

Мы автоматом попадем на ветку master. Но, как мы помним, в этой ветке коммитить небезопасно, как и в ветке dev. Поэтому переходим в Branch – New Branch (Рис. 22).

Консоль:

```
git checkout master  
git checkout dev  
git checkout student  
git pull origin dev
```

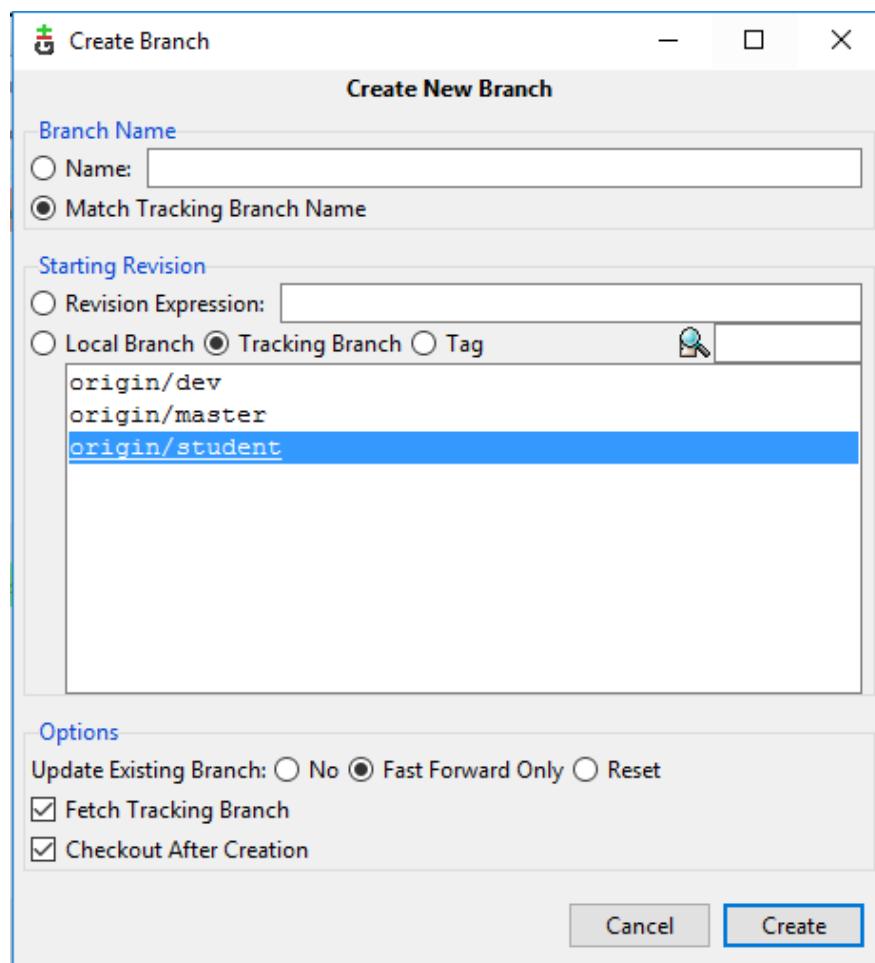


Рисунок 22. Переходим в Branch – New Branch

Вместо Local Branch ставим Tracking Branch, таким образом мы перейдем на локальную ветку, которая связана с удаленной. Если же мы оставим Local Branch, то создастся обычная локальная ветка.

Итак, мы на ветке student, меняем файл main.cpp:

```
#include <iostream>  
using namespace std;  
int main() {
```

```

char name[20];
cout << "Hello world! Have a nice day!\n";
cout << "Enter your name: ";
cin >> name;
cout << "Hello, " << name << "!\n";
cout << "Nice to see you learning Git\n";
system("pause");
return 0;
}

```

Делаем Rescan, добавляем файл в индекс и коммитим с сообщением "fixed namespaces". В консоли это будет так:

```

git add .
git commit -m "fixed namespaces"

```

Далее выполняем уже знакомые операции: переходим на dev, делаем merge с веткой student. Далее аналогично влияем изменения из dev в master

```

git checkout dev
git merge student
git checkout master
git merge dev

```

После этого осталось только синхронизировать удаленный репозиторий с локальным. Для этого мы добавим удаленный репозиторий в список чтобы дальше было удобнее с ним работать. Выбираем Remote – Add (Рис. 23).

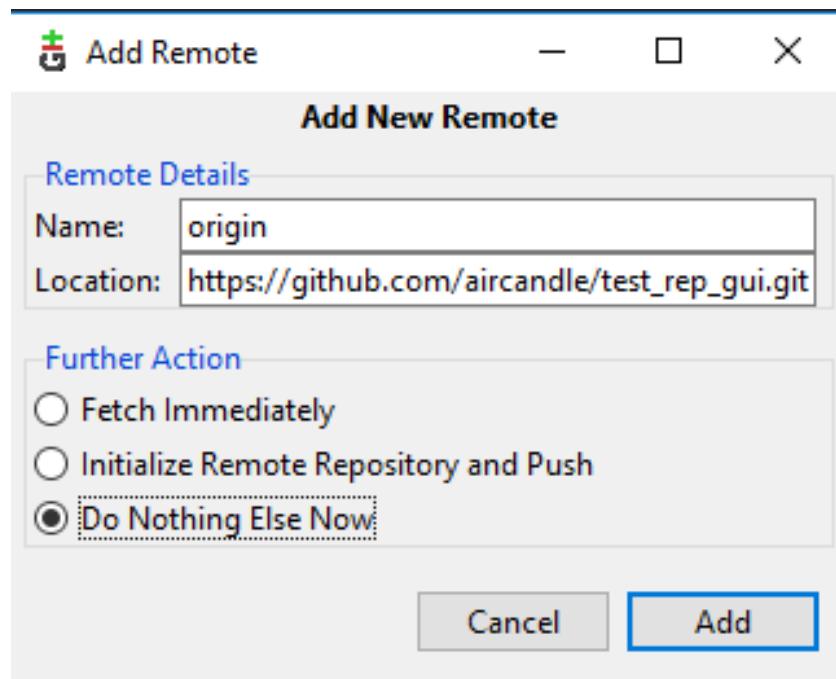


Рисунок 22. Синхронизация локального и удаленного репозиториев

В поле Name вводим сокращенное название для нашего репозитория. Поскольку, как уже говорилось ранее, Git GUI является лишь оболочкой для консольной версии, то

Name используется для более удобного и быстрого способа выполнить команду git push (в консольной версии мы ввели название репозитория на этапе git remote add). Название можно выбрать любое, но негласным стандартом считается название origin. В поле Location вставляем ссылку к нашему удаленному репозиторию, выбираем пункт Do Nothing Else Now и жмем кнопку Add.

Теперь выполняем Remote – Push, выделяем все ветки и жмем Push (рис. 23).

Консоль **git push**

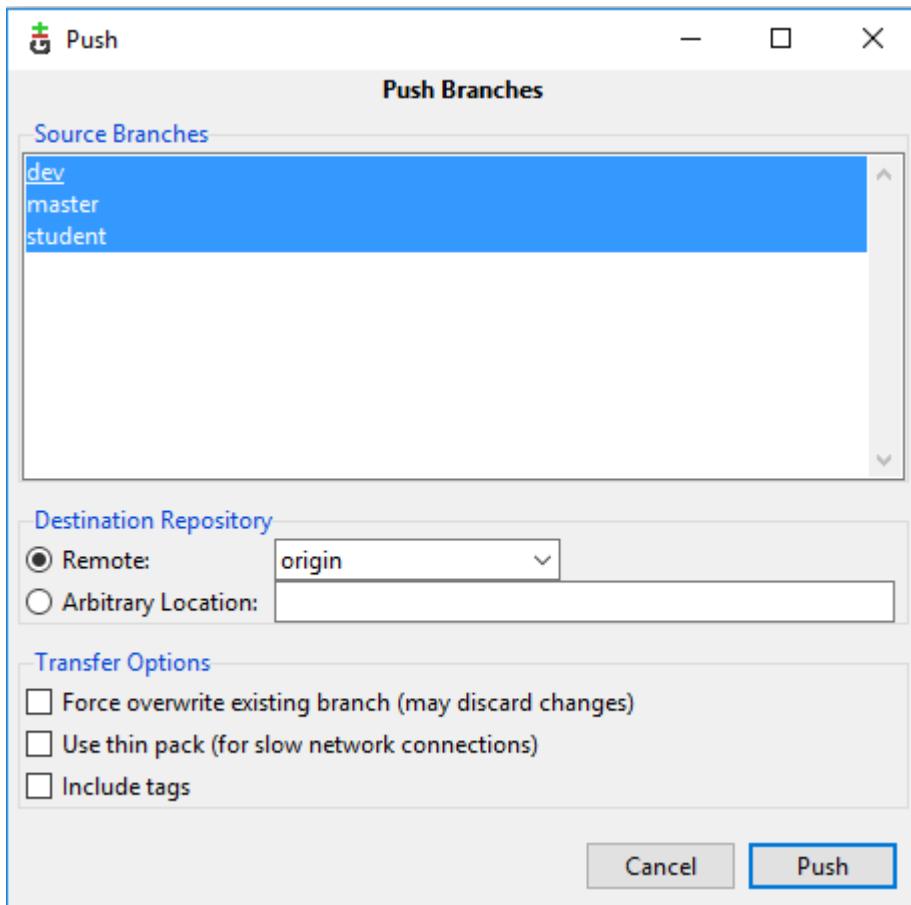


Рисунок 23. Синхронизация локального и удаленного репозиториев

Изменения вылиты на сервер. Посмотрим итоговое дерево коммитов (Рис. 24).

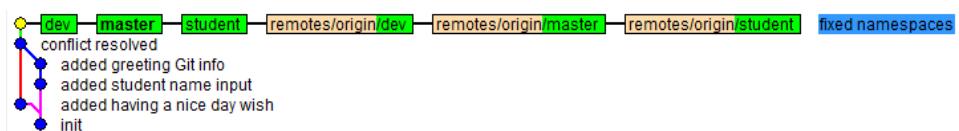


Рисунок 24. Синхронизация локального и удаленного репозиториев

Все ветки находятся в одном месте и синхронизированы между собой.  
На рис. 25 показано как выглядит дерево коммитов на консоли.

The screenshot shows a terminal window titled 'MINGW32:/c/Users/Svan/Desktop/lab5/pull\_bash'. It displays a list of Git commits. The commits are color-coded by author: yellow for the first two, green for the next three, and blue for the last two. The commits are as follows:

- commit 63185818d6de89373d0081def945f5294918e9ab  
Author: Student <yourmail@yourmail.ru>  
Date: Sat Oct 1 20:40:54 2016 +0300  
fixed namespaces
- commit 2657bd9c12374761305336563527da6d369e7f2a  
Merge: 788909a b3cb1f6  
Author: Student <yourmail@yourmail.ru>  
Date: Sat Oct 1 18:55:42 2016 +0300  
conflict resolved
- commit b3cb1f6e9708d51aeac6cb0a0601e5e437a9c7a3  
Author: Student <yourmail@yourmail.ru>  
Date: Sat Oct 1 18:28:22 2016 +0300  
added greeting Git info
- commit a18ed7457a96da8c6b1fe8c8078d68ce0382102b  
Author: Student <yourmail@yourmail.ru>  
Date: Sat Oct 1 18:20:51 2016 +0300  
added student name input
- commit 788909ad38a97254b458c428ddf34b6d61464c18  
Author: Student <yourmail@yourmail.ru>  
Date: Sat Oct 1 18:39:12 2016 +0300  
added having a nice day wish
- commit 7549aa14504ee90d6303a1dd499743ec9444bba9  
Author: Student <yourmail@yourmail.ru>  
Date: Sat Oct 1 18:05:15 2016 +0300

Рисунок 24. Вид дерева коммитов на консоли

*P.S. Если вы захотите в дальнейшей разработке использовать графический клиент для Git, то лучше использовать другие клиенты, нежели Git GUI. Например, достаточно удобны клиенты GitHub for Windows/Mac или же SourceTree. Git GUI для полноценной разработки не самое удобное решение, без возможности решать конфликты, к тому же некоторый функционал консольной версии в нем не реализован, например git pull. Кстати говоря о консоли, в некоторых случаях, например, для клонирования существующего репозитория, консольный git понятнее и удобнее. Поэтому рекомендуется использовать графические клиенты главным образом для решения конфликтов и наглядности при слиянии веток.*

#### **4. Порядок выполнения лабораторной работы.**

1. Составить перечень технологий, используемых для реализации варианта задания
2. Обосновать выбор данных технологий и удобство их использования
3. Декомпозировать разрабатываемую систему, описать модули, необходимые для реализации ПО согласно заданию.
4. Сформулировать набор задач, необходимых для реализации ПО согласно варианту задания, определить порядок выполнения и приоритет каждой из них.
5. Если система контроля версий Git не установлена, то установить ее (параметры оставить по умолчанию).
6. Запустить Git GUI (или консоль). Создать новый репозиторий. Добавить в папку репозитория файлы. Зафиксировать состояние репозитория (выполнить commit).
7. Внести изменения в файлы. Зафиксировать новое состояние репозитория.
8. Создать новую ветку 1. Внести в нее изменения и зафиксировать их.
9. Переключиться на ветку мастера. Внести в нее изменения и зафиксировать их.

10. Продемонстрировать слияние веток.
11. Просмотреть дерево изменений веток (историю).
12. Создать удаленный репозиторий.
13. Загрузить на него свой проект.
14. Обеспечить доступ к нему всем членам команды.
15. Скачать каждым членом команды файлы с удаленного репозитория, произвести изменения и совершить слияние.

Защита лабораторной работы заключается в предъявлении преподавателю полученных результатов (на экране монитора), демонстрации полученных навыков и ответах на вопросы преподавателя.

## **5. Вопросы.**

## **ЛАБОРАТОРНАЯ РАБОТА № 4.**

### **Процессы жизненного цикла программных средств**

**Цель работы:** Изучить различия процессов жизненного цикла (ЖЦ) ПО с точки зрения различных международных и национальных стандартов. Определить целесообразную для выполнения проекта модель ЖЦ. Выполнить технологический процесс кодирования ПО по выполняемому проекту.

**Продолжительность работы – 4 часа.**

#### **Содержание**

1. ЖЦ ПО.....	73
2. Группа стандартов ISO .....	74
3. Группа стандартов IEEE .....	77
4. Национальный стандарт Российской Федерации .....	78
5. Порядок выполнения лабораторной работы.....	84
6. Вопросы.....	84

#### **1. Жизненный цикл программного обеспечения**

Для корректного с точки зрения инженерии и экономики рассмотрения вопросов создания сложных систем необходимо, чтобы был проведен анализ потребностей пользователей, были затронуты вопросы эксплуатации системы, внесения в нее изменений, разработаны функции, удовлетворяющие эти потребности. Без этого невозможно обеспечить, с одной стороны, реальную эффективность системы в виде отношения полученных результатов ко всем сделанным затратам и, с другой стороны, правильно оценивать в ходе разработки степень соответствия системы реальным нуждам пользователей и заказчиков.

Жизненным циклом программного обеспечения называют период от момента появления идеи создания некоторого программного обеспечения до момента завершения его поддержки фирмой-разработчиком или фирмой, выполняющей сопровождение. В ходе жизненного цикла ПО проходит через анализ предметной области, сбор требований, проектирование, кодирование, тестирование, сопровождение и др.

При этом создаются и перерабатываются различного рода модели предметной области, описание требований, техническое задание, архитектура системы, проектная документация на систему в целом и на ее компоненты, прототипы системы и компонентов, собственно, исходный код, пользовательская документация, документация администратора системы, руководство по развертыванию, база пользовательских запросов, план проекта, и пр.

На различных этапах в создание и эксплуатацию ПО вовлекаются люди, выполняющие различные роли. Каждая роль может быть охарактеризована как абстрактная *группа заинтересованных лиц*, участвующих в деятельности по созданию и эксплуатации системы и решают одни и те же задачи или имеющих одни и те же интересы по отношению к ней. Примерами ролей являются: бизнес-аналитик, инженер по требованиям, архитектор, проектировщик пользовательского интерфейса, программист-кодировщик, технический писатель, тестировщик, руководитель проекта по разработке, работник отдела продаж, конечный пользователь, администратор системы, инженер по поддержке и т.п.

Общепринятой структуры жизненного цикла ПО не существует, поскольку она существенно зависит от целей, для которых это ПО разрабатывается и от решаемых им задач. Тем не менее, часто определяют основные элементы структуры жизненного цикла в виде *модели жизненного цикла ПО*. Модель жизненного цикла ПО выделяет конкретные наборы видов деятельности (обычно разбиваемых на еще более мелкие активности), артефактов, ролей и их взаимосвязи, а также дает рекомендации по организации процесса в целом. Эти рекомендации включают в себя ответы на вопросы о том, какие артефакты являются входными данными у каких видов деятельности, а какие появляются в качестве результатов, какие роли вовлечены в различные деятельности, как различные деятельности связаны друг с другом, каковы критерии качества полученных результатов, как оценить степень соответствия различных артефактов общим задачам проекта и когда можно переходить от одной деятельности к другой.

Существует набор стандартов, определяющих различные элементы в структуре жизненных циклов ПО и программно-аппаратных систем. В качестве основных таких элементов выделяют *технологические процессы* — структурированные наборы деятельности, решающих некоторую общую задачу или связанные совокупность задач, такие, как процесс сопровождения ПО, процесс обеспечения качества, процесс разработки документации и пр. Процессы могут определять разные этапы жизненного цикла и увязывать их с различными видами деятельности, артефактами и ролями заинтересованных лиц. Процессы могут разбиваться на подпроцессы, решающие частные подзадачи той задачи, с которой работает общий процесс.

Состав процессов жизненного цикла регламентируется рядом международных и отечественных стандартов, описывающим технологические процессы создания ПО.

Чтобы получить представление о возможной структуре жизненного цикла ПО, обратимся сначала к соответствующим стандартам. Международными организациями, и некоторыми национальными и региональными институтами и организациями такими, как:

- IEEE — Institute of Electrical and Electronic Engineers — Институт инженеров по электротехнике и электронике;
- ISO — International Standards Organization — Международная организация по стандартизации;
- EIA — Electronic Industry Association — Ассоциация электронной промышленности;
- IEC — International Electrotechnical Commission — Международная комиссия по электротехнике;
- ANSI — American National Standards Institute — Американский национальный институт стандартов;
- SEI — Software Engineering Institut — Институт программной инженерии;
- ECMA — European Computer Manufactures Association — Европейская ассоциация производителей компьютерного оборудования
- разработан набор стандартов, регламентирующих различные аспекты жизненного цикла и вовлеченных в него процессов.

### 2.3.1. Группа стандартов ISO

*ISO/IEC 12207 Standard for Information Technology — Software Life Cycle Processes* (процессы жизненного цикла ПО, есть его российский аналог ГОСТ Р ИСО/МЭК 12207-2010). Определяет общую структуру жизненного цикла ПО в виде 3-х ступенчатой модели, состоящей из процессов, видов деятельности и задач. Стандарт

описывает вводимые элементы в терминах их целей и результатов, тем самым задавая неявно возможные взаимосвязи между ними, но не определяя четко структуру этих связей, возможную организацию элементов в рамках проекта и метрики, по которым можно было бы отслеживать ход работ и их результативность.

Самыми крупными элементами являются *процессы жизненного цикла ПО*. Всего выделено 18 процессов, которые объединены в 4 группы (Таблица 1).

Таблица 1. Процессы жизненного цикла ПО по ISO 12207

Основные процессы	Поддерживающие процессы	Организационные процессы	Адаптация
Приобретение ПО; Передача ПО в использование; Разработка ПО; Эксплуатация ПО; Поддержка ПО	Документирование; Управление конфигурациями; Обеспечение качества; Верификация; Валидация; Совместные экспертизы; Аудит; Разрешение проблем	Управление проектом; Управление инфраструктурой; Усовершенствование процессов; Управление персоналом	Адаптация описываемых стандартом процессов под нужды конкретного проекта

Процессы строятся из отдельных *видов деятельности*. Стандартом определены 74 вида деятельности, связанной с разработкой и поддержкой ПО. Здесь мы упомянем только некоторые из них.

- Приобретение ПО включает такие деятельности, как инициация приобретения, подготовка запроса предложений, подготовка контракта, анализ поставщиков, получение ПО и завершение приобретения.
- Разработка ПО включает развертывание процесса разработки, анализ системных требований, проектирование (программно-аппаратной) системы в целом, анализ требований к ПО, проектирование архитектуры ПО, детальное проектирование, кодирование и отладочное тестирование, интеграцию ПО, квалификационное тестирование ПО, системную интеграцию, квалификационное тестирование системы, развертывание (установку или инсталляцию) ПО, поддержку процесса получения ПО.
- Поддержка ПО включает развертывание процесса поддержки, анализ возникающих проблем и необходимых изменений, внесение изменений, экспертизу и передачу измененного ПО, перенос ПО с одной платформы на другую, изъятие ПО из эксплуатации.
- Управление проектом включает запуск проекта и определение его рамок, планирование, выполнение проекта и надзор за его выполнением, экспертизу и оценку проекта, свертывание проекта.
- Каждый вид деятельности нацелен на решение одной или нескольких задач. Всего определено 224 различные задачи. Например:
  - Разворачивание процесса разработки состоит из определения модели жизненного цикла, документирования и контроля результатов отдельных работ, выбора используемых стандартов, языков и инструментов и пр.

– Перенос ПО между платформами состоит из разработки плана переноса, оповещения пользователей, выполнения анализа произведенных действий и пр.

*ISO/IEC 15288 Standard for Systems Engineering — System Life Cycle Processes* [20] (процессы жизненного цикла систем). Отличается от предыдущего нацеленностью на рассмотрение программно-аппаратных систем в целом. В данный момент продолжается работа по приведению этого стандарта в соответствие с предыдущим. ISO/IEC 15288 предлагает похожую схему рассмотрения жизненного цикла системы в виде набора процессов. Каждый процесс описывается набором его *результатов*, которые достигаются при помощи различных видов деятельности.

Всего выделено 26 процессов, объединяемых в 5 групп (Таблица 2).

Таблица 2. Процессы жизненного цикла систем по ISO 15288.

Процессы выработки соглашений	Процессы уровня организации	Процессы уровня проекта	Технические процессы	Специальные процессы
Приобретение системы; Поставка системы	Управление окружением; Управление инвестициями; Управление процессами; Управление ресурсами; Управление качеством	Планирование; Оценивание; Мониторинг; Управление рисками; Управление конфигурацией; Управление информацией; Выработка решений	Определение требований; Анализ требований; Проектирование архитектуры; Реализация; Интеграция; Верификация; Валидация; Передача в использование; Эксплуатация; Поддержка; Изъятие из эксплуатации	Адаптация описываемых стандартом процессов под нужды конкретного проекта

Помимо процессов, определено 123 различных результата и 208 видов деятельности, нацеленных на их достижение. Например, определение требований имеет следующие результаты: Должны быть поставлены технические задачи, которые предстоит решить; должны быть сформулированы системные требования.

Основные деятельности в рамках этого процесса следующие:

- Определение границ функциональности системы;
- Определение функций, которые необходимо поддерживать;
- Определение критериев оценки качества при использовании системы;
- Анализ и выделение требований по безопасности;
- Анализ требований защищенности;
- Выделение критических для системы аспектов качества и требований к ним;
- Анализ целостности системных требований;
- Демонстрация прослеживаемости требований;
- Фиксация и поддержка набора системных требований.

*ISO/IEC 15504 (SPICE) Standard for Information Technology — Software Process Assessment* [21] (оценка процессов разработки и поддержки ПО). Определяет правила оценки процессов жизненного цикла ПО и их возможностей, опирается на модель СММІ (см. ниже) и больше ориентирован на оценку процессов и возможностей их улучшения. В качестве основы для оценки процессов определяет некоторую базовую модель, аналогичную двум описанным выше. В ней выделены категории процессов, процессы и виды деятельности. Определяются 5 категорий, включающих 35 процессов и 201 вид деятельности.

### **2.3.2. Группа стандартов IEEE**

*IEEE 1074-1997 — IEEE Standard for Developing Software Life Cycle Processes* [22] (стандарт на создание процессов жизненного цикла ПО). Нацелен на описание того, как создать специализированный процесс разработки в рамках конкретного проекта. Описывает ограничения, которым должен удовлетворять любой такой процесс, и, в частности, общую структуру процесса разработки. В рамках этой структуры определяет основные виды деятельности, выполняемых в этих процессах и документы, требующиеся на входе и возникающие на выходе этих деятельности. Всего рассматриваются 5 подпроцессов, 17 групп деятельности и 65 видов деятельности.

*IEEE/EIA 12207-1997 — IEEE/EIA Standard: Industry Implementation of International Standard ISO/IEC 12207:1995 Software Life Cycle Processes* [23] (промышленное использование стандарта ISO/IEC 12207 на процессы жизненного цикла ПО). Аналог ISO/IEC 12207, сменил ранее использовавшиеся стандарты J-Std-016-1995 EIA/IEEE Interim Standard for Information Technology — Software Life Cycle Processes — Software Development Acquirer-Supplier Agreement (промежуточный стандарт на процессы жизненного цикла ПО и соглашения между поставщиком и заказчиком ПО) и стандарт министерства обороны США MIL-STD-498.

Например, подпроцесс разработки ПО состоит из групп деятельности по выделению требований, по проектированию и по реализации. Группа деятельности по проектированию включает архитектурное проектирование, проектирование баз данных, проектирование интерфейсов, детальное проектирование компонентов.

В том числе, предусматривается оформление проектной и эксплуатационной документации, подготовка материалов, необходимых для проверки работоспособности и соответствия качества программных продуктов, материалов, необходимых для обучения персонала, и т. д.

По стандарту процесс разработки включает следующие действия:

- *подготовительную работу* – выбор модели жизненного цикла (см. далее), стандартов, методов и средств разработки, а также составление плана работ;
- *анализ требований к системе* – определение ее функциональных возможностей, пользовательских требований, требований к надежности и безопасности, требований к внешним интерфейсам и т. д.;
- *проектирование архитектуры системы* – определение состава необходимого оборудования, программного обеспечения и операций, выполняемых обслуживающим персоналом;
- *анализ требований к программному обеспечению* – определение функциональных возможностей, включая характеристики производительности, среды функционирования компонентов, внешних интерфейсов, спецификаций надежности и безопасности,

эргономических требований, требований к используемым данным, установке, приемке, пользовательской документации, эксплуатации и сопровождению;

- *проектирование архитектуры программного обеспечения* – определение структуры программного обеспечения, документирование интерфейсов его компонентов, разработку предварительной версии пользовательской документации, а также требований к тестам и плана интеграции;
- *детальное проектирование программного обеспечения* – подробное описание компонентов программного обеспечения и интерфейсов между ними, обновление пользовательской документации, разработка и документирование требований к тестам и плана тестирования компонентов программного обеспечения, обновление плана интеграции компонентов;
- *кодирование и тестирование программного обеспечения* – разработку и документирование каждого компонента, а также совокупности тестовых процедур и данных для их тестирования, тестирование компонентов, обновление пользовательской документации, обновление плана интеграции программного обеспечения;
- *интеграцию программного обеспечения* – сборку программных компонентов в соответствии с планом интеграции и тестирование программного обеспечения на соответствие квалификационным требованиям, представляющих собой набор критериев или условий, которые необходимо выполнить, чтобы квалифицировать программный продукт, как соответствующий своим спецификациям и готовый к использованию в заданных условиях эксплуатации;
- *квалификационное тестирование программного обеспечения* – тестирование программного обеспечения в присутствии заказчика для демонстрации его соответствия требованиям и готовности к эксплуатации; при этом проверяется также готовность и полнота технической и пользовательской документации
  - *интеграцию системы* – сборку всех компонентов системы, включая программное обеспечение и оборудование;
  - *квалификационное тестирование системы* – тестирование системы на соответствие требованиям к ней и проверка оформления и полноты документации;
  - *установку программного обеспечения* – установку программного обеспечения на оборудовании заказчика и проверку его работоспособности;
  - *приемку программного обеспечения* – оценку результатов квалификационного тестирования программного обеспечения и системы в целом и документирование результатов оценки совместно с заказчиком, окончательную передачу программного обеспечения заказчику.

### **2.3.3. Национальный стандарт Российской Федерации**

ГОСТ Р ИСО/МЭК 12207-2010. Информационная технология. Системная и программная инженерия. Процессы жизненного цикла программных средств. Настоящий стандарт идентичен международному стандарту ИСО/МЭК 12207-2008\* "Системная и программная инженерия. Процессы жизненного цикла программных средств" (ISO/IEC 12207:2008 "System and software engineering – Software life cycle processes"). Утвержден и введен в действие Приказом Федерального агентства по техническому регулированию и метрологии от 30 ноября 2010 г. N 631-ст.

Данный стандарт, используя устоявшуюся терминологию, устанавливает общую структуру процессов жизненного цикла программных средств, на которую можно ориентироваться в программной индустрии. Он определяет процессы, виды деятельности и задачи, которые используются при приобретении программного

продукта или услуги, при поставке, разработке, применении по назначению, сопровождении и прекращении применения программных продуктов.

Стандарт *не детализирует* процессы жизненного цикла в *терминах методов или процедур*, необходимых для удовлетворения требований и достижения результатов процесса.

Стандарт *не устанавливает* требований к документации в части ее наименований, форматов, определенного содержания и носителей для записи. Стандарт *не устанавливает конкретной модели* жизненного цикла системы или программных средств, разработки методологии, методов, моделей или технических приемов. Стороны, применяющие настоящий стандарт, отвечают за выбор модели жизненного цикла для программных проектов и отображение процессов, действий и задач, представленных в настоящем стандарте, на эту модель. Стороны также ответственны за выбор и применение методов разработки программных средств и за выполнение действий и задач, подходящих для программного проекта.

*Основное требование:* стандарт ГОСТ Р ИСО/МЭК 12207-2010 не должен противоречить политикам, процедурам и нормам применяющей его организации, национальным законам и регулирующим документам. Каждое такое противоречие должно быть разрешено до начала использования настоящего стандарта.

ГОСТ Р ИСО/МЭК 12207-2010 группирует различные виды деятельности, которые могут выполняться в течение жизненного цикла программных систем, в семь групп процессов. Каждый из процессов жизненного цикла в пределах этих групп описывается в терминах цели и желаемых выходов, списков действий и задач, которые необходимо выполнять для достижения этих результатов:

- а) процессы соглашения – два процесса;
- б) процессы организационного обеспечения проекта – пять процессов;
- с) процессы проекта – семь процессов;
- д) технические процессы – одиннадцать процессов;
- е) процессы реализации программных средств – семь процессов;
- ф) процессы поддержки программных средств – восемь процессов;
- г) процессы повторного применения программных средств – три процесса.

Группы процессов жизненного цикла по ГОСТ Р ИСО/МЭК 12207-2010 представлены на рисунке 1.

Эталонная модель процесса не представляет конкретного подхода к осуществлению процесса, как и не предопределяет модель жизненного цикла системы (программного средства), методологию или технологию. Вместо этого эталонная модель предназначается для принятия организацией и базируется на деловых потребностях организации и области приложений. Определенный организацией процесс принимается в проектах организации в контексте требований заказчиков.

Результаты процесса используются для демонстрации успешного достижения цели процесса, что помогает оценщикам процесса определять возможности реализованного процесса организации и предоставлять исходные материалы для планирования улучшений организационных процессов.

Если в предыдущих стандартах выделялись исключительно этапы разработки ПО, то новый стандарт учитывает влияние времени и в нем много внимания уделяется вопросам менеджмента и взаимоотношений покупателей и продавцов ПО, обеспечению качества, вопросам поддержки функционирования ПО и изъятию его из обращения после завершения поддержки ПО разработчиком (Рис.1).

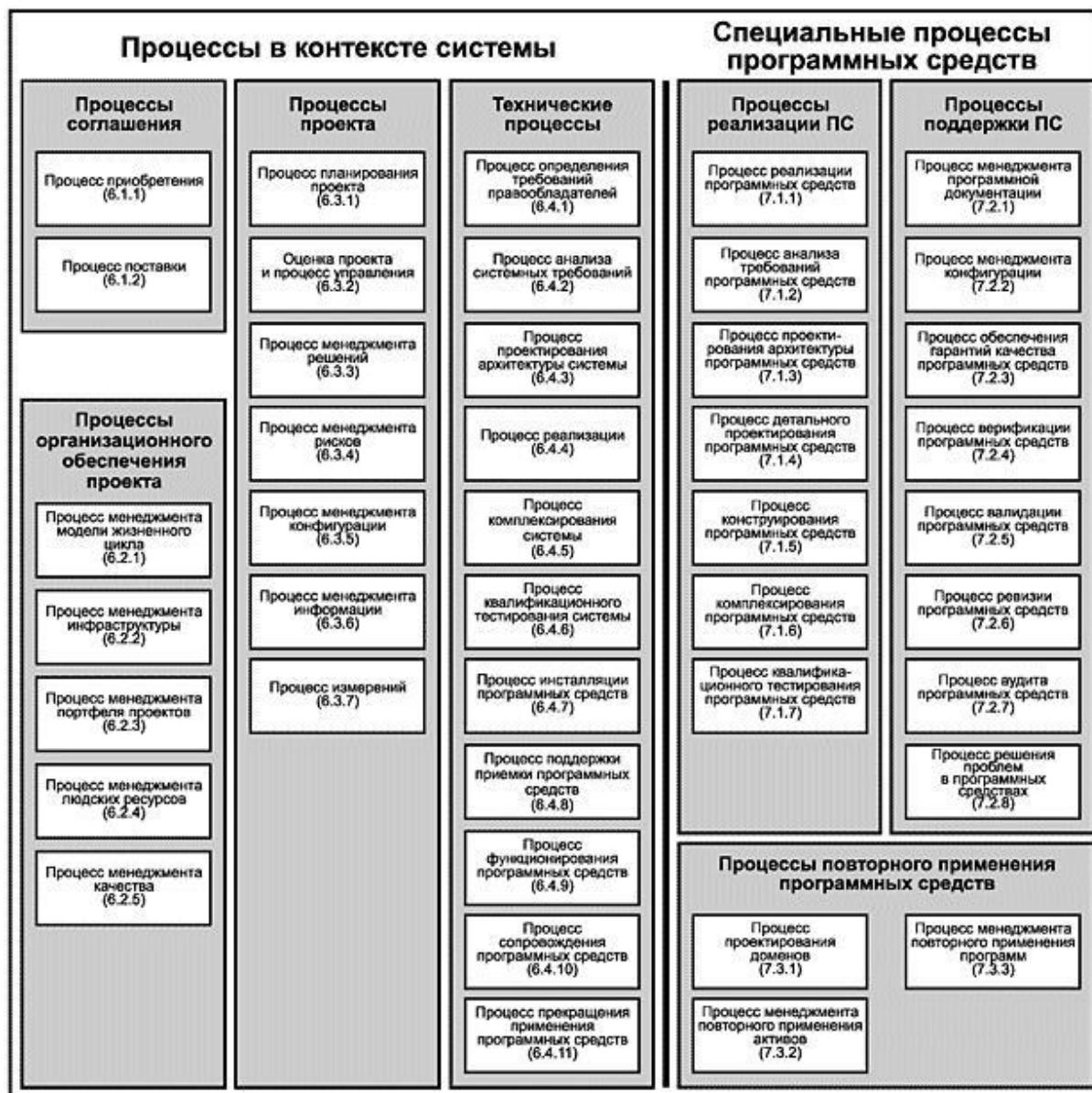


Рисунок 1. Группы процессов жизненного цикла по ГОСТ Р ИСО/МЭК 12207-2010

В соответствии с ГОСТ 19.102-77 выделялись 4 основные этапы разработки ПО:

- постановка задачи (стадия «Техническое задание»);
- анализ требований и разработка спецификаций (стадия «Эскизный проект»);
- проектирование (стадия «Технический проект»);
- реализация (стадия «Рабочий проект»).

Новый стандарт (*ГОСТ Р ИСО/МЭК 12207-2010*) устанавливает 11 процессов в последовательности выполнения работы над проектом:

- a) определение требований правообладателей;
- b) анализ системных требований;
- c) проектирование архитектуры системы;
- d) процесс реализации;
- e) процесс комплексирования системы;
- f) процесс квалификационного тестирования системы;
- g) процесс инсталляции программных средств;

- h) процесс поддержки приемки программных средств;
- i) процесс функционирования программных средств;
- j) процесс сопровождения программных средств;
- k) процесс изъятия из обращения программных средств.

Принципиальное отличие от предыдущего стандарта – пункты c,d, f, j, k.

Стадия *определения требований* к ПО – одна из важнейших и определяет в значительной степени успех всего проекта. Она включает следующие этапы:

- Планирование работ, предваряющее работы над проектом. Определение целей разработки, предварительная экономическая оценка проекта, построение плана-графика выполнения работ, создание и обучение рабочей группы. Иногда вводят дополнительно начальную стадию – анализ осуществимости системы.
- Проведение обследования объекта разработки, в рамках которого осуществляются предварительное выявление требований к будущей системе, определение структуры, определение перечня целевых функций, анализ распределения функций по подразделениям и сотрудникам, выявление функциональных взаимодействий между подразделениями, информационных потоков внутри подразделений и между ними, внешних по отношению к организации объектов и внешних информационных воздействий, анализ существующих средств автоматизации деятельности.
- Построение модели объекта исследования "AS-IS" ("как есть"), отражающей имеющиеся на момент обследования аналогичные решения, выявить узкие места и сформулировать предложения по улучшению ситуации; построение модели "TO-BE" ("как должно быть"), отражающей представление о новых технологиях. Результатом завершения стадии формирования требований к ПО являются спецификации ПО, функциональные, технические и интерфейсные спецификации, для которых подтверждена их полнота, проверяемость и осуществимость.

Стадия *проектирования* включает следующие этапы.

- Разработка системного проекта ПО. На этом этапе дается ответ на вопрос "Что должна делать будущая система?", а именно: определяются архитектура системы, ее функции, внешние условия функционирования, интерфейсы и распределение функций между пользователями и системой, требования к программным и информационным компонентам, состав исполнителей и сроки разработки, план отладки ПО и контроль качества. Основу системного проекта составляют модели проектируемой системы, которые строятся на модели "TO-BE". Результатом разработки системного проекта должна быть одобренная и подтвержденная спецификация требований к ПО.

- Разработка детального (технического) проекта. На этом этапе осуществляется собственно проектирование ПО, включающее проектирование архитектуры системы и детальное проектирование. Таким образом, дается ответ на вопрос: "Как построить систему, чтобы она удовлетворяла требованиям?" Результатом детального проектирования является разработка верифицированной спецификации ПО, включающей: формирование иерархии программных компонентов, межмодульных интерфейсов по данным и управлению; спецификация каждого компонента ПО, имени, назначения, предположений, размеров, последовательности вызовов, входных и выходных данных, ошибочных выходов, алгоритмов и логических схем; формирование физической и логической структур данных до уровня отдельных полей; разработку плана распределения вычислительных ресурсов (времени центральных процессоров, памяти и др.); верификацию полноты, непротиворечивости, осуществимости и обоснованности требований; предварительный план комплексирования и отладки, план руководства для пользователей и приемных испытаний. Завершением стадии

детального проектирования является сквозной контроль проекта или критический поблочный анализ проекта.

Стадия реализации – выполнение следующих работ:

- Разработка верифицированной детальной спецификации каждой подпрограммы (блока не более чем из 100 исходных команд языка высокого уровня). Внешние спецификации должны содержать следующие сведения:

1. *имя модуля* — указывается имя, применяемое для вызова модуля (для модуля с несколькими входами для каждого входа должны быть отдельные спецификации);
  2. *функция* – дается определение функции или функций, выполняемых модулем;
  3. список параметров (число и порядок следования), передаваемых модулю;
  4. *входные и выходные параметры* – точное описание всех данных, принимаемых и возвращаемых модулем (должно быть определено поведение модуля при любых входных условиях);
  5. *внешние эффекты* (печать сообщения, чтение запроса с терминала и т. п.).
- Проектирование логики модулей и программирование (кодирование) модулей.
  - Проверка правильности модулей.
  - Тестирование модулей.
  - Описание базы данных до уровня отдельных параметров, символов и битов.
  - План приемных испытаний.
  - Предварительный план комплексирования и отладки.

Цель процесса комплексирования системы заключается в объединении системных элементов (включая составные части технических и программных средств, ручные операции и другие системы, при необходимости) для производства полной системы, которая будет удовлетворять системному проекту и ожиданиям заказчика, выраженным в системных требованиях. Агрегированные части должны быть проверены, так как они разрабатываются в соответствии со своими требованиями. Процесс комплексирования должен быть задокументирован.

*Квалификационное тестирование* – тестирование, проводимое разработчиком и санкционированное приобретающей стороной (при необходимости) с целью демонстрации того, что программный продукт удовлетворяет спецификациям и готов для применения в заданном окружении или интеграции с системой, для которой он предназначен. Должна обеспечиваться гарантия того, что реализация каждого требования к программным средствам тестируется на соответствие. Результаты квалификационного тестирования должны быть документально оформлены.

Инсталляция программных средств состоит из решения следующих задач:

- разработка плана инсталляции программного продукта в среду его применения, как определено в контракте. Ресурсы и информация, необходимые для инсталляции программного продукта, должны быть определены и быть в наличии. Если инсталлируемый программный продукт заменяет существующую систему, то исполнитель должен поддерживать любые параллельно выполняемые действия, которые требуются в соответствии с контрактом. *План инсталляции должен быть документирован*. Важной частью разработки стратегии инсталляции является наличие стратегии возврата к последней рабочей версии системы.
- Необходимо гарантировать, что базы данных и программный код инициализируются, выполняются и отменяются, как установлено в контракте. События, произошедшие при инсталляции, и их результаты должны документироваться.

Процесс *поддержки приемки* программных средств дополняет выходы процесса передачи и процесса валидации. Цель процесса поддержки приемки программных средств заключается в содействии приобретающей стороне в обеспечении уверенности в том, что продукт соответствует заданным требованиям. В результате успешного осуществления процесса поддержки приемки программных средств:

- продукт комплектуется и поставляется приобретающей стороне;
- поддерживаются приемочные тесты и ревизии, проводимые приобретающей стороной;
- продукт применяется по назначению в среде заказчика;
- проблемы, обнаруженные в течение приемки, идентифицируются и передаются ответственным за их решение.

Результаты ревизий и тестирования должны быть документированы.

Цель процесса *функционирования программных средств* заключается в применении программного продукта в предназначенной для него среде и обеспечении поддержки заказчиков программного продукта. В результате успешного осуществления процесса функционирования программных средств:

- определяется стратегия функционирования;
- определяются и оцениваются условия корректного функционирования программных средств в предназначенной для них среде;
- программные средства тестируются и настраиваются в предназначенной для них среде;
- программные средства функционируют в предназначенной для них среде;
- обеспечиваются содействие и консультации заказчикам программных продуктов в соответствии с условиями соглашения.

Риски, возникающие при функционировании продукта, идентифицируют и непрерывно контролируют.

Цель процесса *сопровождения программных средств* заключается в обеспечении эффективной по затратам поддержки поставляемого программного продукта. В результате успешного осуществления процесса сопровождения программных средств:

- разрабатывается стратегия сопровождения для управления модификацией и перемещением программных продуктов согласно стратегии выпусков;
- выявляются воздействия изменений в существующей системе на организацию, операции или интерфейсы;
- по мере необходимости обновляется связанная с изменениями системная и программная документация;
- разрабатываются модифицированные продукты с соответствующими тестами, демонстрирующими, что требования не ставятся под угрозу;
- обновленные продукты помещаются в среду заказчика;
- сведения о модификации системных программных средств доводятся до всех затронутых обновлениями сторон.

Сопровождающая сторона должна разрабатывать, документировать и выполнять планы и процедуры проведения действий и решения задач в рамках процесса сопровождения программных средств.

Процесс *прекращения применения программных средств* состоит в обеспечении завершения существования системного программного объекта. Этот процесс прекращает деятельность организации по поддержке функционирования и сопровождения или деактивирует, демонтирует и удаляет поврежденные программные продукты. В ходе данного процесса происходит уничтожение или сохранение программных элементов системы и связанных с ними продуктов обычным способом в

соответствии с действующим законодательством, соглашениями, организационными ограничениями и требованиями правообладателей. В результате успешного осуществления процесса прекращения применения программных средств:

- определяется стратегия прекращения применения;
- ограничения по прекращению применения служат в качестве входных данных к требованиям;
- системные программные элементы уничтожаются или сохраняются;
- окружающая среда оставляется в согласованном состоянии;
- обеспечивается доступ к записям, хранящим знания о действиях по прекращению применения, и результатам анализа долговременных воздействий.

Все рассмотренные стандарты могут достаточно сильно разойтись с реальной разработкой, если в ней используются новейшие методы автоматизации разработки и сопровождения ПО. Стандарты организаций ISO и IEEE построены на основе имеющегося эмпирического опыта разработки, полученного в рамках распространенных некоторое время назад парадигм и инструментальных средств. Это не значит, что они устарели, поскольку их авторы имеют достаточно хорошее представление о новых методах и технологиях разработки и пытались смотреть вперед. Но при использовании новаторских технологий в создании ПО часть требований стандарта может обеспечиваться совершенно иными способами, чем это предусмотрено в нем, а часть артефактов может отсутствовать в рамках данной технологии, исчезнув внутри автоматизированных процессов.

### **Порядок выполнения лабораторной работы.**

1. Обосновать модель ЖЦ, наиболее подходящую для вашего проекта.
2. Выделить этапы выполнения проекта в соответствии с ЖЦ.
3. Провести кодирование и комплексную отладку ПО.

### **Вопросы**

1. Какие виды жизненного цикла программного обеспечения вы знаете?
2. Понятие процесса. Виды процессов в соответствии с принятыми стандартами.
3. Основные отличия ГОСТ Р ИСО/МЭК 12207-2010 от ГОСТ 19.102-77.

## **ЛАБОРАТОРНАЯ РАБОТА № 5.**

### **Техники тест-дизайна, написание тест-кейсов. Функциональное тестирование методом «черного ящика»**

**Цель работы:** Научиться использовать техники тест-дизайна при написании тестовых сценариев.

**Продолжительность работы – 4 часа.**

#### **Содержание**

1. Тестирование программного обеспечения .....	85
2. Виды тестовых случаев .....	89
3. Документирование тестовых сценариев .....	92
4. Порядок выполнения лабораторной работы.....	97
5. Вопросы.....	97

#### **1. Тестирование программного обеспечения**

Тестирование программного обеспечения – процесс исследования, испытания *программного продукта, имеющий две различные цели: продемонстрировать разработчикам и заказчикам, что программа соответствует требованиям; выявить ситуации, в которых поведение программы является неправильным, нежелательным или не соответствующим спецификации.*

Тестирование – очень важный и трудоемкий этап процесса разработки программного обеспечения, так как правильное тестирование позволяет выявить подавляющее большинство ошибок, допущенных при составлении программ. Процесс тестирования должен охватывать все этапы разработки ПО. Недостаточно выполнить проектирование и кодирование программного обеспечения, необходимо также обеспечить его соответствие требованиям и спецификациям.

Стандарт ISO/IEC 25010:2011 (ГОСТ Р ИСО/МЭК 25010-2015) определяет качество программного обеспечения как степень удовлетворения системой заявленных и подразумеваемых потребностей различных заинтересованных сторон. В соответствии со стандартом модель качества продукта включает восемь характеристик:

- функциональная пригодность;
- уровень производительности;
- совместимость;
- удобство пользования;
- надёжность;
- защищённость;
- сопровождаемость;
- переносимость (мобильность).

Состав и содержание документации, сопутствующей процессу тестирования, определяется стандартом IEEE 829-2008 IEEE Standard for Software and System Test Documentation.

В зависимости от доступа разработчика тестов к исходному коду тестируемой программы различают «тестирование (по стратегии) белого ящика» и «тестирование (по стратегии) чёрного ящика».

#### **Стратегия черного ящика**

В соответствии с ISTQB (International Software Testing Qualifications Board) тестирование черного ящика определяется следующим образом:

- тестирование, как функциональное, так и нефункциональное, не предполагающее знания внутреннего устройства компонента или системы.

- тест-дизайн, основанный на технике черного ящика – процедура написания или выбора тест-кейсов на основе анализа функциональной или нефункциональной спецификации компонента или системы без знания ее внутреннего устройства.

Данный подход является классическим при написании тестовых сценариев. Тестируемая программа для тестировщика – как черный непрозрачный ящик, содержания которого он не видит. Таким образом, мы не имеем представления о структуре и внутреннем устройстве системы. Нужно концентрироваться на том, что программа делает, а не на том, как она это делает.

### **Стратегия белого ящика**

В соответствии с ISTQB тестирование белого ящика определяется следующим образом:

- тестирование, основанное на анализе внутренней структуры компонента или системы;
- тест-дизайн, основанный на технике белого ящика – процедура написания или выбора тест-кейсов на основе анализа внутреннего устройства системы или компонента.

Стратегия Белого ящика включает в себя:

- покрытие операторов (подразумевает выполнение каждого оператора программы, по крайней мере, один раз);
- покрытие решений (необходимо составить такое число тестов, при которых каждое условие в программе примет как истинное значение, так и ложное значение);
- покрытие условий (если после составления тестов у нас останутся не покрытые операторы, то мы должны дополнить свой набор тестов таким образом чтобы каждый оператор выполняется не менее одного раза);
- покрытие решений и условий (необходимо составить тесты так, чтобы результаты каждого условия выполнялись хотя бы один раз, результаты каждого решения так же выполнялись хотя бы один раз, и каждый оператор должен быть выполнен хотя бы один раз);
- комбинаторное покрытие условий (все возможные комбинации результатов условий в каждом решении, а также каждый оператор выполнились по крайней мере один раз).

Таблица 1. Сравнение тестирования методом черного и белого ящика

Критерий	Стратегия черного ящика	Стратегия белого ящика
Уровни, к которым применима техника	В основном: Системное тестирование Приемочное тестирование	В основном: Юнит-тестирование Интеграционное тестирование
Кто выполняет	Как правило, тестировщик	Как правило, разработчик
Знание программирования	Не нужно	Необходимо
Знание реализации	Не нужно	Необходимо
Основа для тест-кейсов	Спецификация, требования	Проектная документация

## **Стратегия серого ящика**

Стратегия серого ящика – комбинация методов белого ящика и чёрного ящика, состоящая в том, что к части кода и архитектуры у тестировщика доступ есть, а к части – нет. Явное упоминание стратегии серого ящика – крайне редкий случай: обычно говорят о методах белого или чёрного ящика в применении к тем или иным частям приложения, при этом понимая, что «приложение целиком» тестируется по методу серого ящика.

Надо понимать, что правильно выполненное Unit-тестирование всегда выполняется с применением стратегии серого ящика (подробно в лабораторной работе №6).

### **Функциональное тестирование методом «чёрного ящика»**

*Функциональное тестирование* объекта-тестирования планируется и проводится на основе требований к тестированию, заданных на этапе определения требований. Цель функциональных тестов состоит в том, чтобы проверить соответствие приложения установленным требованиям.

Основным источником исходных данных для тестирования являются функциональные и нефункциональные требования к разрабатываемой системе. Эти требования могут создаваться в различной форме. Например, это может быть Техническое Задание (по любому из ГОСТов) или артефакты RUP (документ VISION – Концепция, спецификации прецедентов и т.д.).

В основе функционального тестирования лежит методика "чёрного ящика". Идея тестирования сводится к тому, что группа тестировщиков проводит тестирование, не имея доступа к исходным текстам тестируемого приложения. При этом во внимание принимается только входящие требования и соответствие им тестируемым приложением.

При тестировании "чёрного ящика" (black box) программа рассматривается как объект, внутренняя структура которого неизвестна. Тестировщик вводит данные и анализирует результат, но, как именно работает программа, он не знает. Подбирая тесты, специалист ищет интересные с его точки зрения входные данные и условия, которые могут привести к нестандартным результатам. Интересны для него прежде всего те представители каждого класса входных данных, на которых с наибольшей вероятностью могут проявиться ошибки тестируемой программы.

Тестирование «чёрного ящика» обеспечивает поиск следующих категорий ошибок:

- некорректных или отсутствующих функций;
- ошибок интерфейса;
- ошибок во внешних структурах данных или в доступе к внешней базе данных;
- ошибок инициализации и завершения.

Полное тестирование, когда проверяются все возможные варианты выполнения программы, нереально. Поэтому тестирование должно базироваться на некотором подмножестве всевозможных тестовых сценариев. Существует различные методики выбора этого подмножества.

## **Тестовые артефакты**

В соответствие с процессами или методологиями разработки ПО, во время проведения тестирования создается и используется определенное количество тестовых

артефактов (документы, модели и т.д.). Наиболее распространенными тестовыми артефактами являются:

- **План тестирования** (Test Plan) – это документ описывающий весь объем работ по тестированию, начиная с описания объекта, стратегии, расписания, критериев начала и окончания тестирования, до необходимого в процессе работы оборудования, специальных знаний, а также оценки рисков с вариантами их разрешения.
- **Набор тест кейсов и тестов** (Test Case & Test suite) – это последовательность действий, по которой можно проверить соответствует ли тестируемая функция установленным требованиям.
- **Дефекты / Баг Репорты** (Bug Reports / Defects) – это документы, описывающие ситуацию или последовательность действий приведшую к некорректной работе объекта тестирования, с указанием причин и ожидаемого результата.

Тестовая документация бывает двух видов: **внешняя и внутренняя**. И та, и другая – инструмент, облегчающий жизнь проектной команде. Не более и не менее.

#### **Внешняя документация:**

- Замечание – короткая записка, комментарий о небольшой неточности в реализации продукта.
- Баг-репорт – описание выявленного случая несоответствия производимого продукта требованиям, к нему выдвигаемым – ошибки или ее проявление. Он обязательно должен содержать следующие элементы:
  1. Идею тестового случая, вызвавшего ошибку.
  2. Описание исходного состояния системы для выполнения кейса.
  3. Шаги, необходимые для того, чтобы выявить ошибку или ее проявление.
  4. Ожидаемый результат, т. е. то, что должно было произойти в соответствии с требованиями.
  5. Фактический результат, т. е. то, что произошло на самом деле.
  6. Входные данные, которые использовались во время воспроизведения кейса.
  7. Прочую информацию, без которой повторить кейс не получится.
  8. Критичность и/или приоритет.
  9. Экранный снимок (скрин).
  10. Версию, сборку, ресурс и другие данные об окружении.
- Запрос на изменение (улучшение) – описание неявных/некритичных косвенных требований, которые не были учтены при планировании/реализации продукта, но несоблюдение, которых может вызвать неприятие у конечного потребителя. И пути/рекомендации по модификации продукта для соответствия им.
- Отчет о тестировании (тест репорт) – документ, предоставляющий сведения о соответствии/ несоответствии продукта требованиям. Может так же содержать описание некоторых подробностей проведенной сессии тестирования, например, затраченное время, использованные виды тестирования, перечень проверенных случаев и т. п. В идеальном варианте фраза вида «Тест пройден. Ошибка не воспроизводится/Функционал работает корректно/Соответствует требованиям» означает, что продукт или его часть полностью соответствует требованиям прямым и косвенным (в производстве ПО).

#### **Внутренняя документация:**

- *Тест-план (план тестирования)* – формализованное и укрупненное описание одной сессии тестирования по одному или нескольким направлениям проверок. Т.е.

перечень направлений проверок, которые должны быть проведены в рамках сессии тестирования (и, сообразных этим направлениям, требований). Также может содержать в себе необходимую информацию об окружении, методике, прочих условиях важных для показательности данной сессии тестирования. Под направлением проверок также может пониматься более детализированная тестовая документация (в виде ссылки на нее): чек листы, тестовые комплекты, тестовые сценарии, на которую необходимо опираться при проведении сессии тестирования. Основная цель документа – описать границы сессии тестирования, стабилизировать показательность данной сессии.

- *Тестовый сценарий* – последовательность действий над продуктом, которые связаны единым ограниченным бизнес-процессом использования, и сообразных им проверок корректности поведения продукта в ходе этих действий. Может содержать информацию об исходном состоянии продукта для запуска сценария, входных данных и прочие сведения, имеющие определяющее значение для успешного и показательного проведения проверок по сценарию. Особенностью является линейность действий и проверок, т.е. зависимость последующих действий и проверок от успешности предыдущих. Цель документа – стабилизация покрытия аспектов продукта, необходимых для выполнения функциональной задачи, показательными необходимыми и достаточными проверками. Фактически при успешном прохождении всего тестового сценария мы можем сделать заключение о том, что продукт может выполнять ту или иную возложенную на него функцию.

- *Тестовый комплект* – некоторый набор формализованных тестовых случаев объединенных между собой по общему логическому признаку.

- *Чек-лист (лист проверок)* – перечень формализованных тестовых случаев в виде удобном для проведения проверок. Тестовые случаи в чек-листе не должны быть зависимыми друг от друга. Обязательно должен содержать в себе информацию о: идеях проверок, наборах входных данных, ожидаемых результатах, булевую отметку о прохождении/непрохождении тестового случая, булевую отметку о совпадении/несовпадении фактического и ожидаемого результата по каждой проверке. Может так же содержать шаги для проведения проверки, данные об особенностях окружения и прочую информацию необходимую для проведения проверок. Цель – обеспечить стабильность покрытия требований проверками необходимыми и достаточными для заключения о соответствии им продукта. Особенностью является то, что чек-листы компонуются теми тестовыми случаями, которые показательны для определенного требования.

- *Тестовый случай (тест-кейс)* – формализованное описание одной показательной проверки на соответствие требованиям прямым или косвенным. Обязательно должен содержать следующую информацию:

1. Идея проверки.
2. Описание проверяемого требования или проверяемой части требования.
3. Используемое для проверки тестовое окружение.
4. Исходное состояние продукта перед началом проверки.
5. Шаги для приведения продукта в состояние, подлежащее проверке.
6. Входные данные для использования при воспроизведении шагов.
7. Ожидаемый результат.
8. Прочую информацию, необходимую для проведения проверки.

**Тестовый случай (Test Case)** – это артефакт, описывающий совокупность шагов, конкретных условий и параметров, необходимых для проверки реализации тестируемой функции или её части.

Под тест кейсом понимается структура вида:  
**Действие > ожидаемый результат > результат теста**

Пример:

Действие	ожидаемый результат	результат теста (выполнено/не выполнено)
Открыть страницу "login"	страница отрывается	выполнено

## 2. Виды Тестовых Случаев

Тест кейсы разделяются по ожидаемому результату на позитивные и негативные:

**Позитивный тест кейс** использует только корректные данные и проверяет, что приложение правильно выполнило вызываемую функцию.

**Негативный тест кейс** оперирует как корректными так и некорректными данными (минимум 1 некорректный параметр) и ставит целью проверку исключительных ситуаций (срабатывание валидаторов), а также проверяет, что вызываемая приложением функция не выполняется при срабатывании валидатора.

### Техники тест-дизайна

Рассмотрим программу, которая должна принять шесть символов на вход и убедиться, что первый символ является числовым и а остальные буквенно-цифровыми. Посчитаем только количество вариантов тестов с позитивным результатом: 10 цифр, 62 алфавитных символов (верхнего и нижнего регистра) и 10 цифр составляют  $(10 * 62^5) = 9\ 161\ 328\ 320$  вариантов. Как вывод, полное тестирование (exhaustive testing, complete testing) невозможно.

Вместо исчерпывающего тестирования, мы используем анализ рисков и расстановку приоритетов и техники тест дизайна. Процесс тест-дизайна включает в себя анализ и трансформацию имеющейся документации и требований в тест-кейсы и чек-листы.

Выделяют следующие техники тест-дизайна:

- Эквивалентное разделение;
- Анализ граничных значений;
- Причинно-следственный анализ;
- Предугадывание ошибки;
- Исчерпывающее тестирование.

### Разбиение на классы эквивалентности

Одними из ключевых понятий теории тестирования являются классы эквивалентности и граничные условия. Разбиение на классы эквивалентности представляет собой технологию проектирования тестов, ориентированную на снижение общего числа тестов, необходимых для подтверждения корректности функциональных возможностей программы.

*Класс эквивалентности* – набор тестовых данных с общими свойствами. Обрабатывая разные элементы класса, программа ведет себя одинаково. Если один из тестов выявит ошибку, остальные, скорее всего, тоже это сделают и наоборот.

Класс эквивалентности может задавать набор допустимых или недопустимых значений. Нельзя забывать о классах, охватывающих заведомо неверные или недопустимые входные данные.

#### **Правила формирования классов эквивалентности.**

1. Если условие ввода задает диапазон  $n..m$ , определяется один допустимый и два недопустимых класса эквивалентности

Допустимый класс –  $[n..m]$

Недопустимый класс –  $x < n$  – меньше нижней границы

Недопустимый класс –  $x > m$  – больше верхней границы

2. Если условие ввода задает конкретное значение  $a$ , то определяется один допустимый и два недопустимых класса эквивалентности

Допустимый класс  $\{a\}$

Недопустимый класс  $x < a$

Недопустимый класс  $x > a$

3. Если условие задает множество значений  $\{a,b,c\}$ , то определяются один допустимый и один недопустимый класс эквивалентности.

Допустимый класс  $\{a,b,c\}$

Недопустимый класс  $x: (x \neq a) \& (x \neq b) \& (x \neq c)$

4. Если условие задает булево значение, то определяется два класса эквивалентности:  $\{\text{true}\}$ ,  $\{\text{false}\}$ .

5. Следует также выявлять группы переменных, совместно участвующих в определенных вычислениях, результат которых ограничивается конкретным набором или диапазоном значений

#### **Анализ граничных значений**

Анализ граничных значений представляет собой технологию проектирования тестов, которая является дополнением разбиения на классы эквивалентности. Вместо того, чтобы выбирать некоторый конкретный элемент класса эквивалентности, анализ граничных значений предлагает проектировщику выбрать элементы, которые находятся на границе класса. Как правило, большая часть ошибок происходит на границах области ввода, а не в центре.

Неправильные операторы сравнения (например,  $>$  вместо  $\geq$ ) вызывают ошибки только на граничных значениях аргументов. В то же время программа, которая сбоят на промежуточных значениях диапазона, почти наверняка будет сбить и на граничных.

*Анализ граничных значений* заключается в тестировании каждой границы класса эквивалентности, причем с обеих сторон. Программа, которая пройдет эти тесты, скорее всего, пройдет и все остальные, относящиеся к данному классу.

Правила анализа граничных условий.

1. Если условие ввода (вывода) задает диапазон значений, то тестируется

- минимальное и максимальное значение диапазона.
- значения чуть меньше минимума и чуть больше максимума.

2. Если входные или выходные данные являются упорядоченными множествами (например, последовательным файлом, линейным списком, таблицей), то тестируется обработка первого и последнего элементов этих множеств.

#### **Причино-следственный анализ.**

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.**

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.**

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.

## **3. Документирование тестовых сценариев**

*Тестовый сценарий* (test case) – это описание входной информации, условий и последовательности выполнения действий, а также ожидаемого выходного результата.

В силу того, что целью тестирования является выявление дефектов, хорошим тестом считается тест с высокой вероятностью обнаружения дефекта. Чтобы спроектировать такой тест, специалист по тестированию должен встать на позицию «конструктивного разрушения» программного продукта. Конструктивное разрушение означает выявление проблем в программном продукте с целью их устранения.

Учитывая, что даже тесты, не выявившие ошибку, выполняются неоднократно в ходе регрессионного тестирования, наличие описаний тестовых сценариев необходимо. Однако уровень их подробности и формальности может быть разным.

Не существует стандартных промышленных форматов, в которых задаются и отслеживаются тестовые сценарии и примеры. Каждая организация выбирает свой формат записи тестовых сценариев. Единственный критерий заключается в том, что они должны предоставлять идентификаторы, ясное описание шагов теста, тестовых данных и ожидаемых результатов.

- Таким образом, тестовый сценарий должен обладать следующими свойствами:
- высокая вероятность обнаружения дефектов
- воспроизводимость
- наличие четко определенных ожидаемых результатов и критериев успешного или неуспешного выполнения теста
- неизбыточность.

Далее представлено несколько вариантов тестовых сценариев.

## Простой формат тестового сценария

Тестовая спецификация Система: Contact Management System			Page <u>1</u>							
Разработано: André Sissener	Начальные установки: Открыть окно ввода клиента (плана, контакта)									
Источник тестовых данных: Sample Company Info from Sandy	Цели: Проверить функциональность ввода данных и управляющих кнопок									
Test Case #	Описание	Шаги теста	Ожидаемые результаты	Реальные результаты	Прошел / Провал	Тестер/Дата				
1	Тестирование функциональности добавления одной компании	Ввести информацию о компании и сохранить	Информация о компании должна быть сохранена корректно	✓	Прошел	André Sissener 11/21/04				
2	Тестирование функциональности добавления нескольких компаний	Ввести информацию о компании и сохранить, далее добавить еще одну компанию	Информация о компании сохраняется и появляется новый экран ввода	✓	Прошел	André Sissener 11/21/04				
3	Тестирование функциональности добавления плана работы с компанией через форму компании	Ввести информацию о плане работы и связать его с компанией	План должен быть сохранен и связан с правильной компанией	✓	Прошел	André Sissener 11/21/04				
4	Тестирование функциональности добавления нового контакта	Ввести информацию о контакте и сохранить	Информация о контакте должна быть сохранена	✓	Прошел	André Sissener 11/21/04				
5	Тестирование функциональности связывания контакта с компанией и планом	После ввода информации о контакте, кликнуть на компанию и план, с которыми необходимо связать контакт	Компания и план должны быть связаны с компанией	✓	Прошел	André Sissener 11/21/04				
6	Тестирование функциональности ввода плана	Ввести информацию о плане и	План сохраняется без связи с	✓	Прошел	André Sissener 11/21/04				

	контактов без присоединения к контактам	сохранить, не присоединяя к контактам	контактами			
7	Тестирование функциональности ввода контакта без присоединения к компании	Ввести информацию о контакте и сохранить, не присоединяя к компании	Окно сообщения «Вы должны присоединить контакт к компании»	см. результаты в log_ts t246.txt	Провалился	André Sissener 11/21/04

Каждая строка обозначает отдельный тестовый пример. Если тестировщики регистрируют в таблицах достаточно существенную информацию, то программисты могут воспользоваться такими таблицами для точного воспроизведения провалившегося теста.

Другим вариантом документирования тестов является представление в одном документе отдельного уникального тестового сценария.

<b>Идентификация</b>	
ID тестового примера: <u>тп-402</u> Система: <u>Графический редактор</u> Цель: <u>Проверка функции открытия файла</u> Тестер: _____	Версия программы: _____ Операционная система: _____ Дата проведения теста: _____
<b>Начальные установки</b> Запустить приложение	
<b>Шаги теста</b> 1. Выбрать меню «Файл» 2. Выбрать «Открыть» 3. Выбрать файл и нажать ОК	
<b>Ожидаемые результаты</b> 1. Выпадает список команд меню «Файл» 2. Появляется диалоговое окно «Открыть» со списком файлов 3. Выбранный файл отображается в главном окне	
<b>Действительные результаты</b> <input checked="" type="checkbox"/> Пройден <input type="checkbox"/> Провален	

Схема теста дополняется таблицей тестовых входных данных, которая может иметь следующий формат.

Тестовые данные					
	Сценарий 1	Сценарий 2	Сценарий 3	Сценарий 4	Сценарий 5
<b>Значение 1</b>					
<b>Значение 2</b>					
<b>...</b>					

<b>Значение X</b>					
<b>Ожидаемый результат</b>					
<b>Реальный результат (если отличается от ожидаемого)</b>					
<b>Прошел/Провалился</b>					
<b>Окружение (если провалился)</b>					
<b>Log Number (if failed)</b>					

Для приложений, в которых содержится много входных и выходных значений, можно использовать следующую таблицу. Каждая строка будет соответствоватьциальному тестовому примеру, а каждый столбец задает входное значение и ожидаемые результаты.

### Тестовые сценарии на основе прецедентов

Прецеденты – это эффективный механизм документирования требований. Правильно сконструированный прецедент можно легко преобразовать в тестовый пример. Каждый прецедент позволяет генерировать несколько тестовых примеров. Основной успешный сценарий дает информацию для первого тестового примера. Для каждого альтернативного потока также должен существовать, по крайней мере, один тестовый пример.

**Пример.** В системе «Путеводитель» для местного торгового центра хранится список торговых точек, которые относятся к различным категориям (Мужская одежда, Женская одежда, Цветы, Подарки и др). Покупатель может найти нужную торговую точку и получить карту ее местоположения.

#### Прецедент «Поиск магазина».

##### *Основной успешный сценарий.*

1. Покупатель выбирает категорию торговых точек. Система отображает список торговых точек данной категории.

2. Покупатель выбирает торговую точку из списка. Система отображает на экране карту местоположения этой торговой точки.

3. Покупатель выбирает функцию печати. Система выдает печатную копию карты.

##### *Альтернативные потоки.*

1а. Покупатель хочет получить справку при выборе категории торговой точки

1а1. Покупатель нажимает кнопку СПРАВКА. Система отображает экран со справкой, представляющий информацию относительно главного меню.

1б. Нет подходящей категории торговых точек.

1б1. Прецедент завершается.

2а. Покупатель хочет получить справку при выборе названия торговой точки.

2а1. Покупатель нажимает кнопку СПРАВКА. Система отображает экран со справкой, представляющей информацию относительно меню торговых точек.

Данный прецедент преобразуется в следующий тестовый сценарий.

Test Case #	Описание	Шаги теста	Ожидаемые результаты	Реальные результаты	Прошел / Провалился	Тестер Дата
п1-1	прецедент «Поиск-магазина»	1. Выбрать «Мужская одежда» 2. Выбрать название первой торговой точки 3. Нажать кнопку печати	1. Появляется список экранов с торговыми точками и перечисляются два магазина 2. На экране выдается карта с этой торговой точкой 3. Выдается печатная копия карты	✓	Прошел	
п1-2	прецедент «Поиск-магазина» расширение 1а	1. Нажать кнопку СПРАВКА 2. Нажать кнопку ВЕРНУТЬСЯ НАЗАД	1. Появляется экран со справкой, представляющей информацию относительно главного меню 2. Возврат к главному меню	✓	Прошел	
п1-3	прецедент «Поиск-магазина» расширение 1б	[Покупатель ищет книжную торговую точку]	В главном меню отсутствует категория книжных торговых точек	✓	Прошел	
п1-4	прецедент «Поиск-магазина» расширение 2а	1. Выбрать пункт «Рестораны» 2. нажать кнопку СПРАВКА	1. Появляется экран со списком торговых точек, на котором перечислено 5 ресторанов 2. Выдается экран со справкой относительно меню торговых точек	✓	Прошел	

#### **4. Порядок выполнения лабораторной работы.**

1. Написать тестовые сценарии для тестирования вашего проекта. Первый уровень тестирования – наличие ошибок в Use-Case-диаграммах. Следующий уровень тестирования – ваша программа. Если программа не завершена – тестируете прототип.
2. Провести тестирование в соответствии со следующими техниками тест-дизайна:
  - Эквивалентное разделение;
  - Анализ граничных значений;
  - Причинно-следственный анализ;
  - Предугадывание ошибки;
  - Исчерпывающее тестирование.

#### **5. Вопросы.**

1. В чем заключается задача тестирования ПО?
2. Что такое тестовые артефакты?
3. Что такое техники тест-дизайна?
4. Что такое тестовый сценарий?

# **ЛАБОРАТОРНАЯ РАБОТА № 6.**

## **Unit-тестирование.**

**Цель работы:** научиться планировать и разрабатывать модульные тесты.

**Продолжительность работы – 4 часа.**

### **Содержание**

1. Уровни тестирования .....	98
2. Unit-тестирование.....	99
3. Планирование тестов.....	103
4. Организация тестирования .....	104
5. Шаблон написания unit-теста .....	105
6. Классы-дублеры.....	109
7. Тестовое покрытие .....	109
8. Преимущества unit-тестирования .....	111
9. Порядок выполнения лабораторной работы.....	112
10. Вопросы.....	112

### **1. Уровни тестирования**

Тестирование – процесс многогранный, различаемый по самым разным признакам:

#### *1. По объекту тестирования:*

- Функциональное тестирование.
- Тестирование производительности.
- Нагрузочное тестирование.
- Стресс-тестирование.
- Тестирование стабильности.
- Конфигурационное тестирование.
- Юзабилити-тестирование.
- Тестирование интерфейса пользователя.
- Тестирование безопасности.
- Тестирование локализации.
- Тестирование совместимости.

#### *2. По знанию системы.*

- Тестирование чёрного ящика.
- Тестирование белого ящика.
- Тестирование серого ящика.

#### *3. По степени автоматизации.*

- Ручное тестирование.
- Автоматизированное тестирование.
- Полуавтоматизированное тестирование.

#### *4. По времени проведения тестирования.*

- Альфа-тестирование.
- Дымовое тестирование (англ. smoke testing).
- Тестирование новой функции (new feature testing).
- Подтверждающее тестирование.
- Регрессионное тестирование.
- Приёмочное тестирование.
- Бета-тестирование.

*5. По признаку позитивности сценариев.*

Позитивное тестирование.

Негативное тестирование.

*6. По степени подготовленности к тестированию.*

Тестирование по документации (формальное тестирование).

Интуитивное тестирование (англ. ad hoc testing).

По степени изолированности компонентов выделяют четыре уровня тестирования программного обеспечения:

- **Модульное Тестирование (Unit Testing)** — модуль это наименьшая функциональная часть программы или приложения, которая не может функционировать отдельно, а только лишь в сочетании с другими модулями. Тем не менее, после разработки этого модуля мы уже можем приступить к тестированию и найти несоответствия с нашими требованиями. Модульное тестирование заключается в тестировании этого отдельного модуля, как части программы, подразумевая, что это только модуль и он не может существовать самостоятельно и является частью приложения, программы
- **Интеграционное Тестирование (Integration Testing)** — следующий уровень тестирования, который проводится после модульного тестирования. После того как отдельные модули нашего приложения были протестированы, нам следует провести интеграционное тестирование, чтобы убедиться что наши модули успешно функционируют в связке друг с другом. Иными словами тестируем 2 и более связанных модуля, чтобы проверить что интеграция прошла успешно и без явных багов.
- **Системное Тестирование (System Testing)** — уровень тестирования, в котором мы проводим тестирование целой системы или приложения, полностью разработанного и которое уже готово к потенциальному релизу. На этом уровне мы тестируем систему, приложение в целом, проводим тестирования на всех требуемых браузерах или операционных системах (если десктоп приложение) и проводим все требуемые типы тестирования такие как: функциональное, тестирование безопасности, тестирование юзабилити, тестирование производительности, нагрузочное тестирование и т.д.
- **Приемочное Тестирование (Acceptance Testing)** — после успешного завершения системного тестирования, продукт проходит уровень приемочного тестирования, который обычно проводится заказчиком или любыми другими заинтересованными лицами, с целью убеждения, что продукт выглядит и работает так, как требовалось изначально и было описано в требованиях к продукту. Приемочное тестирование также может проводиться после каждого из вышеописанных уровней тестирования

## 2. Unit-тестирование.

**Unit-тесты** — это тестирование одного модуля кода (обычно это одна функция или один класс в случае ООП-кода) в изолированном окружении. Unit-тест потому и называется модульным, что тестирует отдельные модули, а не их взаимодействие.

Причем, чем меньше тестируемый модуль — тем лучше с точки зрения будущей поддержки тестов. Для тестирования *взаимодействия* используются *интеграционные* тесты где тестируются скорее полные use cases, а не отдельная функциональность.

Суть *Unit-тестирования* заключается в проверке на корректность работы классов программы. Сложность задачи состоит в том, что классы часто используют в своей работе другие классы, которые как раз тестиовать и не надо. Что делать, если вызов одного метода влечет за собой цепочку вплоть до базы данных? Основное правило для создания *юнит-теста*: тестируемый код *не должен работать с сетью* (и внешними серверами), файлами, базой данных (иначе мы тестируем не саму функцию или класс, а еще и диск, базу, и т.д.). Как этого добиться, рассмотрим ниже.

Обычно *юнит-тест* передает функции разные входные данные и проверяет, что она вернет ожидаемый результат. Например, если у нас есть функция проверки правильности

номера телефона, мы даем ей заранее подготовленные номера, и проверяем, что она определит их правильно. Если у нас есть функция решения квадратного уравнения, мы проверяем, что она возвращает правильные корни (для этого мы заранее делаем список уравнений с ответами).

*Unit-тесты* хорошо тестируют код, который содержит какую-то логику. Если в коде мало логики, а в основном содержатся обращения к другим классам, то юнит-тесты написать может быть сложно (так как надо сделать замену для других классов и не очень понятно, что именно проверять?).

За годы существования понятия *unit-тестирования* были выработаны некоторые подходы, которые помогают несколько автоматизировать создание тестов. «Автоматизировать» в данном случае не означает, что за вас какая-то программа напишет тесты. Она означает, что есть определенная стратегия написания, которая упрощает работы над созданием тестов.

## Паттерны unit-тестирования

**Паттерн Arrange-act-assert** структурирует код теста в три секции: `arrange`, `act`, `assert`. Схема паттерна представлена на рисунке 1.

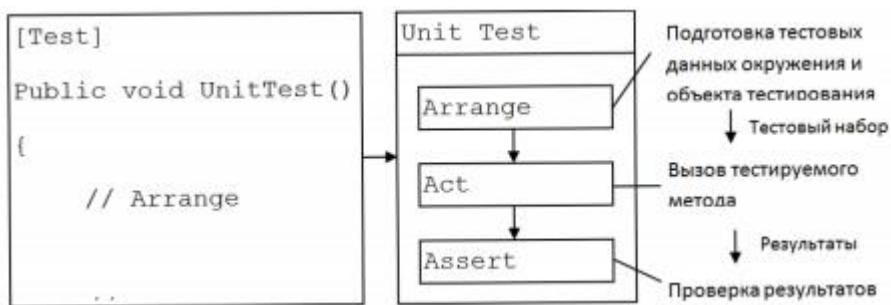


Рис. 1. Структура модульного теста. Паттерн AAA.

Каждый раз при запуске тест должен пройти через 4 стадии.

*Arrange – инициализация.* На самом деле это один из самых сложных этапов. Надо создать для программы полную иллюзию того, что она работает в обычном режиме. А сделать это не так уж и просто. Так при работе с базой данных для заполнения какой-либо таблицы нужен заранее сформированный набор справочных данных. Тех же контрагентов, пользователей, прав и т.п. Откуда это все появится? Надо будет это все придумать.

Нередко бывает, что для метода какого-то класса надо создать еще десяток объектов. Тоже задачка достаточно сложная. Иногда можно пойти по пути создания окружения для каждого теста в отдельности, иногда — для всех тестов вместе.

Например — вы хотите проверить свой сервлет. Но на вход ему надо подать объекты, которые реализуют интерфейсы `HttpServletRequest` и `HttpServletResponse`. Web-сервер их предоставляет, но нам (по основному правилу тестирования) надо исключить его воздействие на программу. Значит все необходимое надо как-то создавать самим.

*Act – запуск.* Необходимо выполнить метод или несколько методов. Если все необходимые данные созданы на этапе инициализации, то здесь особо и говорить не о чем. Правда есть одно "НО" – тесты должны выполняться быстро. Вряд ли вам понравиться, что при запуске тестов надо будет каждый раз идти пить кофе. При такой производительности вас и с работы могут попросить. Поэтому иногда приходится придумывать подходы, которые позволяют выполнять методы быстро. Особенно это важно при обращении к внешним ресурсам – базам данных, внешним службам. Иногда бывает так, что надо обратиться к тому, чего у нас вообще нет – например надо работать с

кассовым аппаратом. Приходится пользоваться разделением интерфейсов и реализаций. Ваша система работает с интерфейсом, за которым «прячется» эмулятор кассового аппарата. Так что если вдруг у вас нет реального объекта для тестирования – заменяйте его фиктивным и тестируйте свою систему с таким. Что же касается базы данных – иногда тестировать надо не саму работу с базой данных, а обработку данных из нее. Можно сделать опять же какой-нибудь заменитель, который будет возвращать нужные нам данные. А это будет быстрее, чем обращение к реальной базе. Особенно если учесть, что таких обращений может быть много.

Здесь можно посоветовать следующее – проектируйте свою программу сразу с учетом того, что надо будет писать тесты.

*Assert – проверка..* На этом шаге просто проверяем, что программа сделала то, что от нее и требовалось. Для этого надо проверить полученные данные, либо как-то отследить действия системы (по логам, по вызовам – заменить вызовы реальные на вызовы подставных объектов). По данным проще, и часто этого хватает. Отслеживание действия – задача более серьезная. Здесь надо предусмотреть обработку разных веток программы, верную реакцию на исключения. А если учесть, что такие исключения бывают не только внутри программы, но и снаружи, от внешних ресурсов – задачка нетривиальная.

*И что особенно важно (кстати этого нет в названии паттерна!) – освобождение ресурсов* – просто сделайте все, чтобы ваши действия вернули систему в состояние до ваших разрушительных действий. Удаляйте данные, освобождайте ресурсы – в общем здесь тоже есть над чем подумать.

Реализация всех этих этапов – задача достаточно не только творческая. Надо копить опыт, придумывать, пробовать.

Пример шаблона для написания теста:

```
[TestMethod]
public void TestLabel()
{
    TestObject example = new TestObject(); // Arrange.
    string result = example.label; // Act.
    Assert.AreEqual("TestObjectLabel", result); // Assert
}
```

**Паттерн Object Mother** представляет собой набор фабричных методов, позволяющих создавать похожие объекты в тестах. В результате создаются объекты в предустановленном состоянии. Каждый раз, когда пользователю требуется другой набор данных, создается новый фабричный метод. Недостатком является увеличение размера материнского объекта.

Пример шаблона для написания теста:

```
// Object Mother
public class TestUsers {
    public static User aRegularUser() {
        return new User("John Smith", "jsmith", "42xcc", "ROLE_USER");
    }
    // other factory methods
}
// arrange
User user = TestUsers.aRegularUser();
User adminUser = TestUsers.anAdmin();
```

**Паттерн Test Data Builder** использует паттерн *Builder (Строитель)* для создания объектов в *unit-тестах*. Данный паттерн позволяет задать лишь те атрибуты, которые необходимы в конкретном тестовом сценарии. Для каждого класса, который используется в teste, создается Builder, обладающий следующими характеристиками:

- имеет экземпляр переменной для каждого параметра конструктора;
- инициализирует экземпляры переменных для часто используемых или безопасных значений;
- имеет метод `build`, позволяющий создавать объекты на основе значений переменных;
- имеет публичный метод для переопределения значений переменных.

Пример шаблона для написания теста:

```
public class UserBuilder {  
    public static final String DEFAULT_NAME = "John Smith";  
    public static final String DEFAULT_ROLE = "ROLE_USER";  
    public static final String DEFAULT_PASSWORD = "42";  
  
    private String username;  
    private String password = DEFAULT_PASSWORD;  
    private String role = DEFAULT_ROLE;  
    private String name = DEFAULT_NAME;  
  
    private UserBuilder() {  
    }  
  
    public static UserBuilder aUser() {  
        return new UserBuilder();  
    }  
  
    public UserBuilder withName(String name) {  
        this.name = name;  
        return this;  
    }  
  
    public UserBuilder withUsername(String username) {  
        this.username = username;  
        return this;  
    }  
  
    public UserBuilder withPassword(String password) {  
        this.password = password;  
        return this;  
    }  
  
    public UserBuilder withNoPassword() {  
        this.password = null;  
        return this;  
    }  
  
    public UserBuilder inUserRole() {  
        this.role = "ROLE_USER";  
        return this;  
    }  
  
    public UserBuilder inAdminRole() {  
        this.role = "ROLE_ADMIN";  
        return this;  
    }  
}
```

```

    }

    public UserBuilder inRole(String role) {
        this.role = role;
        return this;
    }

    public UserBuilder but() {
        return UserBuilder
            .aUser()
            .inRole(role)
            .withName(name)
            .withPassword(password)
            .withUsername(username);
    }

    public User build() {
        return new User(name, username, password, role);
    }
}

```

Вызов в тестах осуществляется следующим образом:

```

UserBuilder userBuilder = UserBuilder.aUser()
    .withName("John Smith")
    .withUsername("jsmith");

User user = userBuilder.build();
User admin = userBuilder.but()
    .withNoPassword().inAdminRole();

```

Такой подход улучшает читаемость кода и устраниет проблему с несколькими фабричными методами для изменения объектов, которые используются в случае *Object Mother*.

### 3. Планирование тестов

Первый вопрос, который встает перед нами: «Сколько нужно тестов». Ответ: тестов должно быть столько, чтобы не осталось неоттестированных участков. Можно даже ввести формальное правило: код с не оттестированными участками не может быть опубликован.

*Проблема состоит в том, что хотя неоттестированный код почти наверняка неработоспособен, но полное покрытие не гарантирует работоспособности.* Написание тестов, исходя только из уже существующего кода только для того, чтобы иметь стопроцентное покрытие кода тестами – порочная практика. Такой подход со всей неизбежностью приведет к существованию оттестированного, но неработоспособного кода. Кроме того, метод белового ящика, как правило, приводит к созданию позитивных тестов. А ошибки, как правило, находятся негативными тестами. В тестировании вопрос «Как я могу сломать?» гораздо эффективней вопроса «Как я могу подтвердить правильность».

В первую очередь тесты должны соответствовать не коду, а требованиям. Правило, которое следует применять:

- Тесты должны базироваться на спецификации.
- Один из эффективных инструментов, для определения полноты тестового набора — матрица покрытия.

- На каждое требование должен быть, как минимум, один тест. Неважно, ручной или автоматический.

При подготовке тестового набора рекомендуется начать с простого позитивного теста. Затраты на его создание минимальны. Да вероятность создания кода, не работающего в штатном режиме, гораздо меньше, чем отсутствие обработки исключительных ситуаций. Но исключительные условия в работе программы редки. Как правило, все работает в штатном режиме. Тесты на обработку некорректных условий, находят ошибки гораздо чаще, но если выяснится, что программа не обрабатывает штатные ситуации, то она просто никому не нужна.

Простой позитивный тест нужен т.к. несмотря на малую вероятность нахождения ошибки, цена пропущенной ошибки чрезмерно высока.

Последующие тесты должны создаваться при помощи формальных методик тестирования. Таких как, классы эквивалентности, исследование граничных условий, метод ортогональных матриц и т.д. (см. лаб. работу №5).

Последнюю проверку полноты тестового набора следует проводить с помощью формальной метрики «Code Coverage». Она показывает неполноту тестового набора. И дальнейшие тесты можно писать на основании анализа неоттестированных участков.

#### 4. Организация тестирования

Тесты (test cases) могут группироваться в наборы (test suites). В случае провала тестирования, программист увидит имена виновников – наборов и тестов. В связи с этим, имя должно сообщать программисту о причинах неприятности и вовлеченных модулях.

Имя может быть полезным и в других случаях. Например, заказчик нашел в программе ошибку, значит программист должен либо добавить новый тест, либо исправить существующий. Следовательно, нужна возможность найти тест по характеру ошибки, а помочь в этом могут хорошо выбранные имена.

Обычно имя теста состоит из наименования тестируемого класса или функции и отражает особенности проверяемого поведения. Например, имена TestIncorrectLogin и TestWeakPasswordRegistration могли бы использоваться для проверки корректности логина и реакции модуля на слишком простой пароль.

Тесты могут располагаться как вместе с тестируемым кодом, так и в отдельном проекте. Очевидно, что основной проект не должен зависеть от тестов, ему незачем знать что его кто-то тестирует. Разделение тестов и рабочего кода считается хорошим тоном, однако бывают исключения – например, если нам надо проверить реализацию, т.е. корректность работы его закрытых (private) методов класса. В таком случае тесты могут размещаться прямо внутри тестируемого кода (или используется отношение дружбы между тестируемым классом и тестом). Обычно такой необходимости не возникает, а тестами покрывают лишь интерфейс, т.к. именно он формирует поведение класса.

Мало того, что основной проект не должен зависеть от тестов – тесты не должны зависеть друг от друга. В противном случае на результаты тестирования может оказывать влияние порядок выполнения тестов (т.е. фаза луны). Из этого простого правила можно сделать следующие важные выводы:

- недопустимо изменять глобальные объекты в тестах и тестируемых модулях. Если же тестируемый код взаимодействует с глобальным объектом (например базой данных), то проверка корректности такого взаимодействия – удел системного тестирования;
- каждый тестовый случай должен быть настроен на выявление лишь одной возможной ошибки – иначе по имени ошибки нельзя установить причину неполадок;
- модули проекта должны соблюдать принцип единой ответственности (Single responsibility principle, SRP) – в противном случае наши тесты будут выявлять более чем по одной ошибке;

- тесты не должны дублировать друг друга. Методология разработки через тестирование (test driven developing, TDD) предполагает короткие итерации, каждая из которых содержит этап рефакторинга (переработки кода). Переработке должен подвергаться не только код основного проекта, но и тесты.

Наконец, юнит-тесты запускаются достаточно часто, поэтому должны работать быстро. В тестовых случаях не должны обрабатываться большие объемы информации – тестовые случаи должны содержать только то, что связано непосредственно с проверяемым поведением. Проверка корректности работы системы на больших объемах данных должна быть вынесена на этап нагружочного тестирования.

## 5. Создание Unit-тестов в Visual studio

Создание unit-тестов рассмотрим на примере тестирования класса Client (Рис. 2).

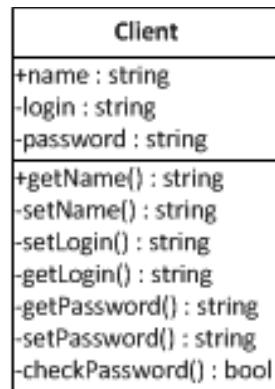


Рисунок 2. Диаграмма класса Client

В Обозревателе решений кликаем правой кнопкой мыши по **Решению (Solution)**. В контекстном меню выбираем **Добавить**, затем **Создать проект** (Рис. 3).

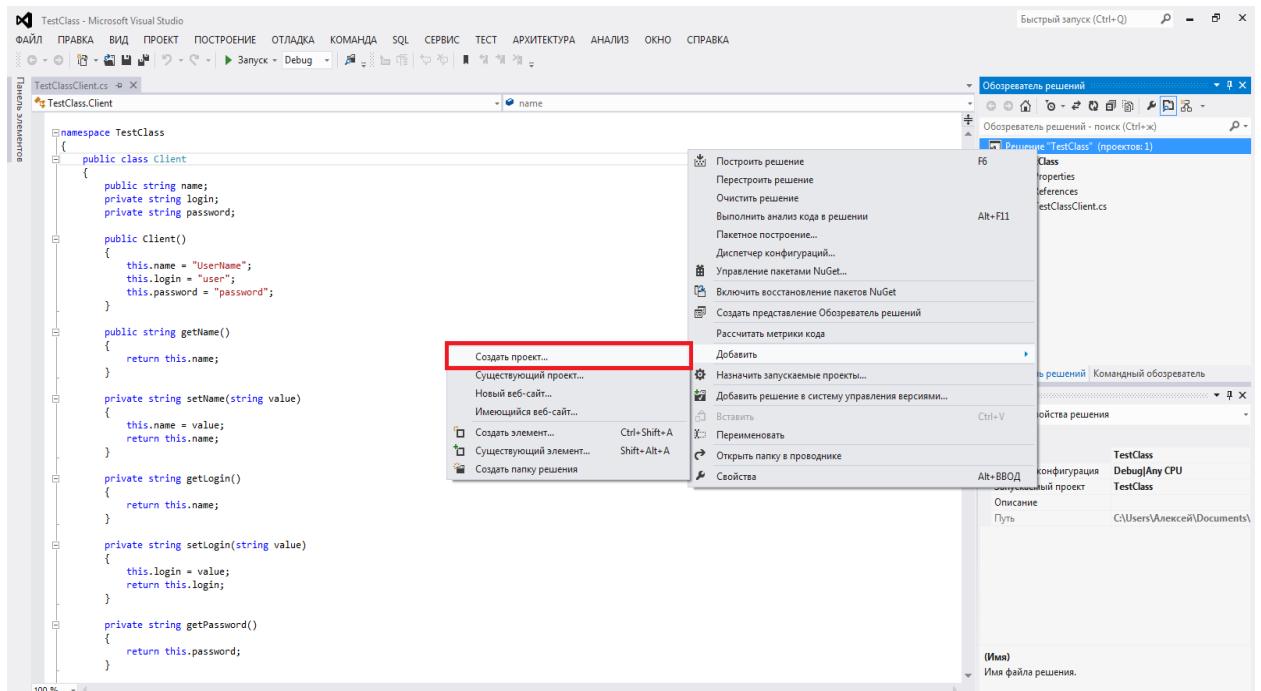


Рисунок 3. Создание проекта

В списке проектов переключаемся на вкладку **Тесты**, выбираем **Проект модульного теста (Unit Test Project)** и задаем имя. Нажимаем OK (Рис. 4).

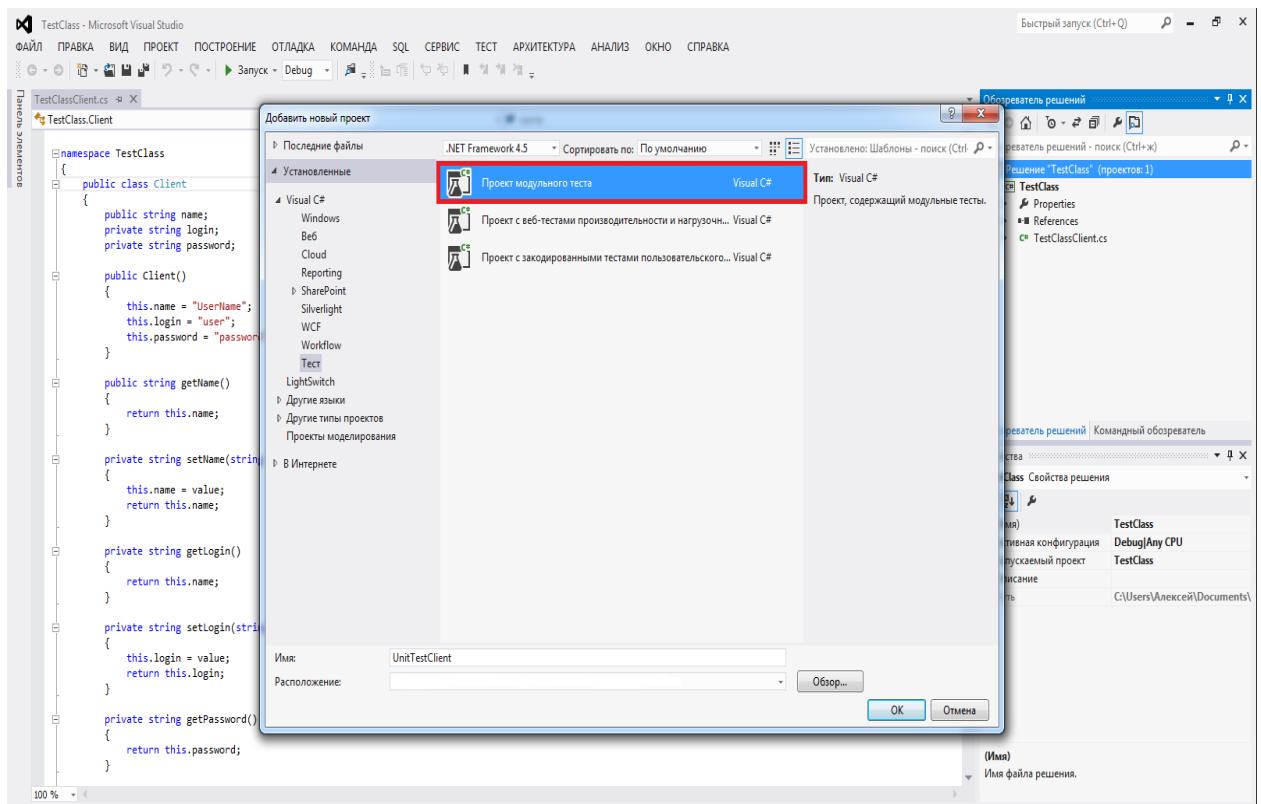


Рисунок 4. Создание проекта модульного теста

В результате создается проект, содержащий следующий сгенерированный код:

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace UnitTestClient
{
    [TestClass]
    public class UnitTest1
    {
        [TestMethod]
        public void TestMethod1()
        {
        }
    }
}
```

*Unit-тестами* являются методы, помеченные атрибутами `[TestMethod]`. *Unit-тесты* собираются в классы, помеченные атрибутом `[TestClass]`.

Подключаем сборку проекта, который необходимо протестировать. Для этого в **Обозревателе решений** кликаем правой кнопкой мыши по зависимостям (**References**) в проекте unit-тестов. В контекстном меню выбираем **Добавить ссылку** (Рис. 5).

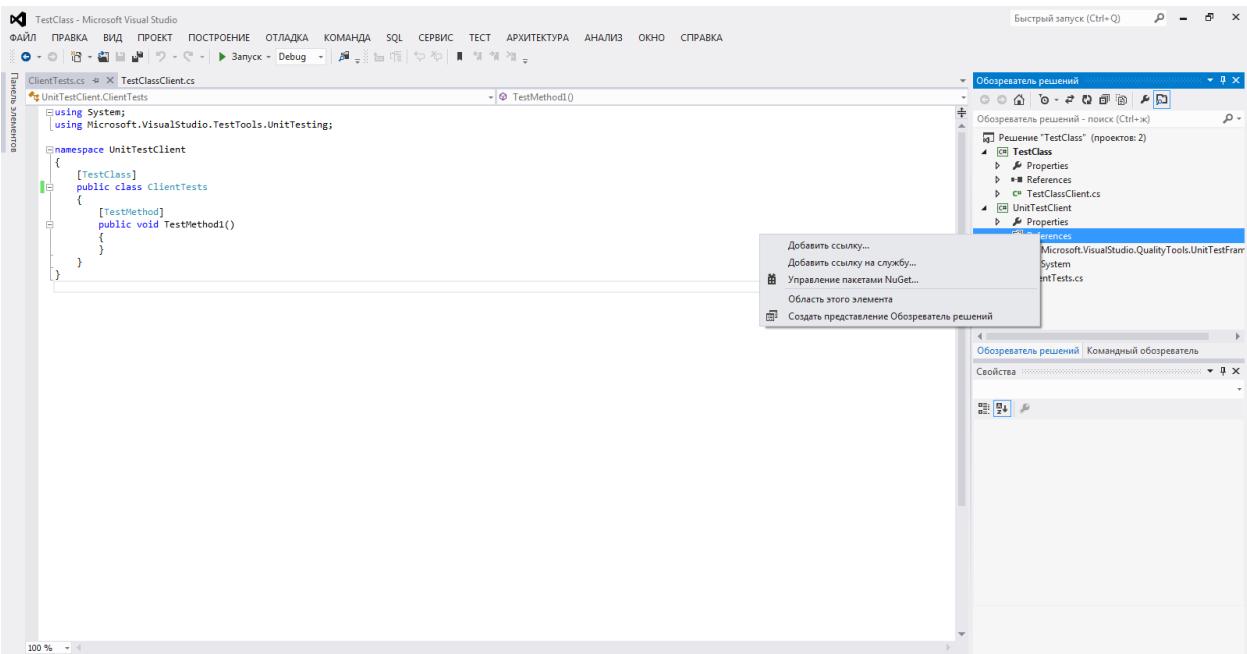


Рисунок 5. Создание проекта модульного теста

Переходим на вкладку **Решение**, выбираем среди проектов соответствующий, поставив галочку рядом с ним. Нажимаем **OK** (Рис. 6).

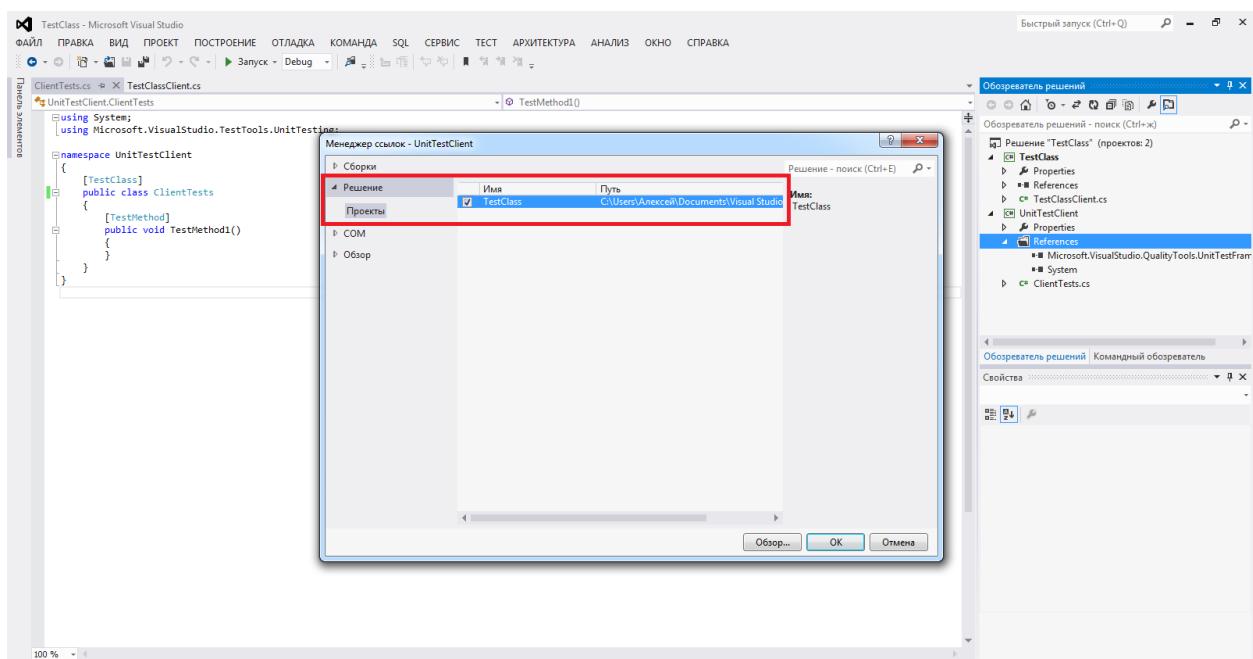


Рисунок 6. Выбор проекта модульного теста

Напишем unit-тест для проверки значения атрибута name при использовании конструктора класса Client:

```
[TestMethod]
public void NameByDefaultConstructor()
{
    string expectedName = "UserName"; //Arrange
    TestClass.Client client = new TestClass.Client(); //Act
```

```

        Assert.AreEqual(expectedName, client.getName()); //Assert
    }
}

```

Вызовем правой кнопкой мыши контекстное меню и выберем **Выполнить тесты** (Рис. 7).

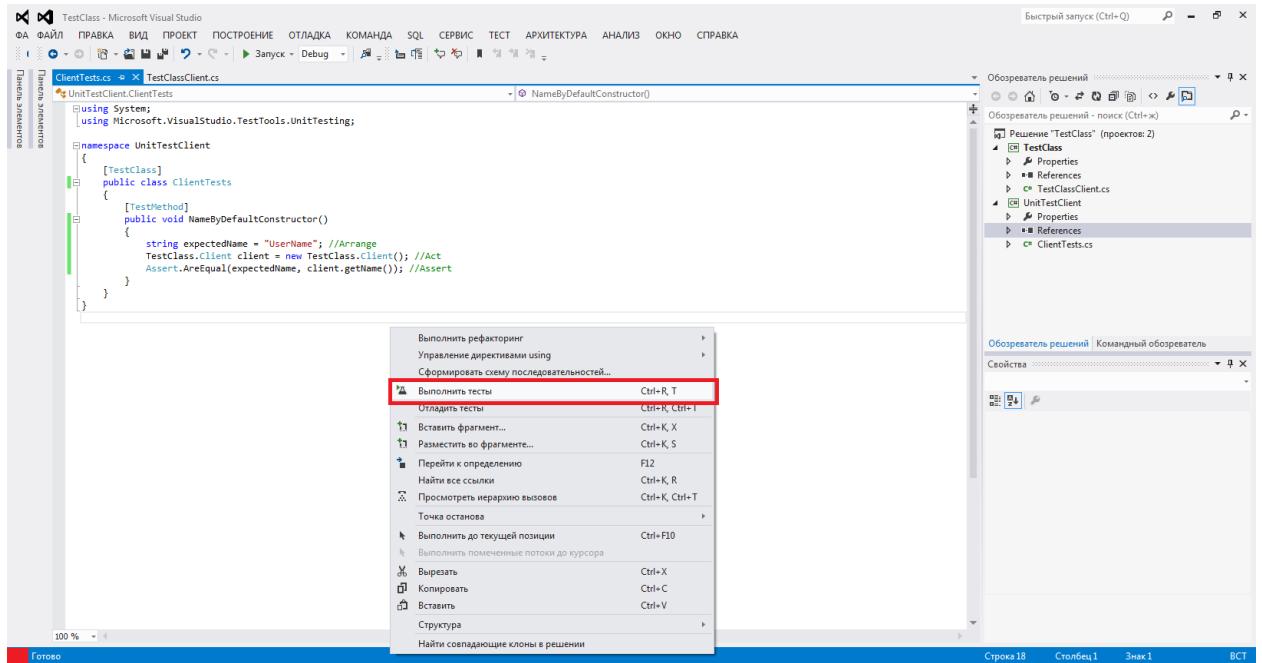


Рисунок 7. Выполнение теста

Тесты начнут выполняться, и откроется окно **Обозревателя тестов**, содержащее информацию о выполнении unit-тестов (результат выполнения, время и т.д.) (Рис. 8).

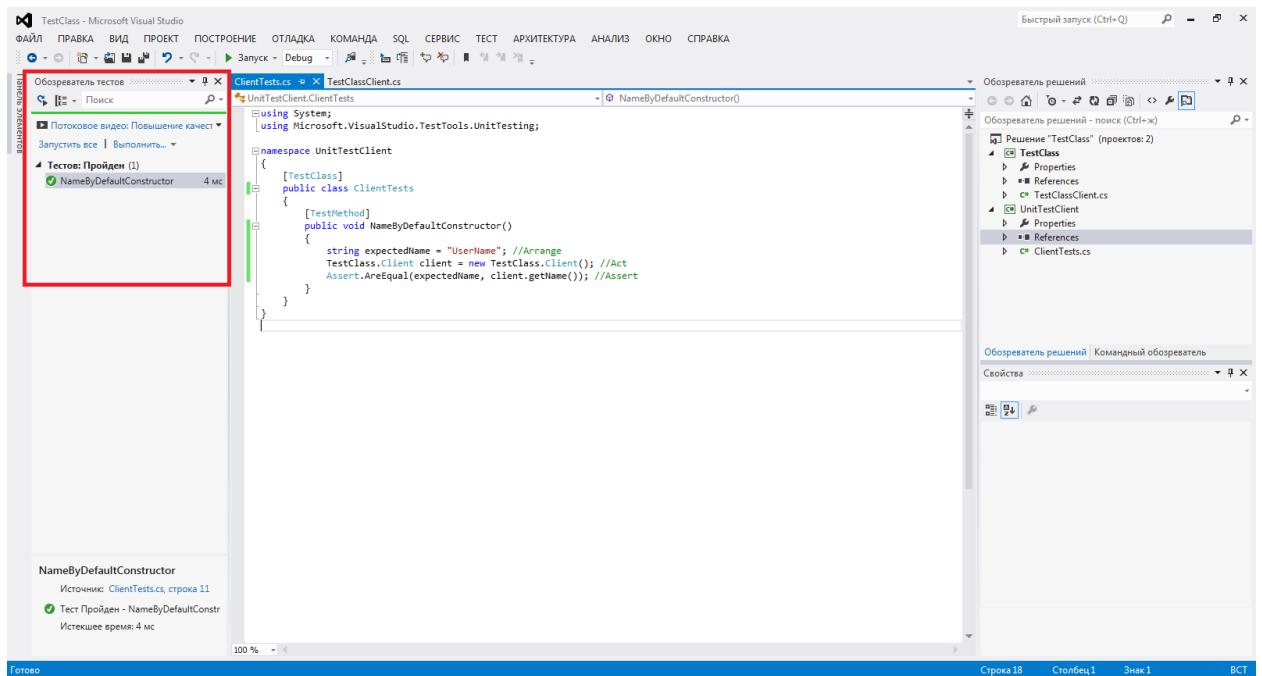


Рисунок 8. Результат выполнение теста

## 6. Виды тестовых объектов

В соответствии с классическим трудом по модульным тестам Жерарда Месароша ("xUnit test patterns: refactoring test code"), вводятся следующие типы тестовых объектов:

- **dummy object** (**муляж**) предназначен для успешного прохождения проверок во время компиляции и во время выполнения. Муляж не принимает участия в тестовом сценарии, он передается в качестве аргументов методов, но никогда не используется. В качестве dummy object может выступать нулевая ссылка (null reference), пустой объект или константа;
- **test stub** (заглушка) используется для получения данных из внешней зависимости, подменяя её. Возвращается предопределенный результат, и при этом игнорируются все данные, которые могут поступать из тестируемого объекта в stub. Например, stub класса работы с базой данных может вместо реального обращения к базе данных возвращать, что запрос успешно выполнен. А при попытке прочитать что-то из нее возвращает заранее подготовленный массив с данными;
- **test spy** (тестовый шпион) осуществляет запись вызовов и данных, поступающих из тестируемого объекта для проверки корректности вызова зависимого объекта. Данный объект предназначен для проверки логики конкретного тестируемого объекта без проверок зависимых объектов;
- **mock object** (мок-объект) частично реализует интерфейс и обеспечивает возможность убедиться, что вызовы мок объекта удовлетворяют спецификации. Мок объект проверяет корректность поведения тестируемого объекта и имеет предопределенное поведение в рамках конкретного тестового сценария. Они также выступают в качестве своего рода записывающего устройства: они отслеживают, какие из методов имитации объекта вызываются, с какими параметрами и сколько раз;
- **fake object** (фальшивый объект) представляет собой замену тяжеловесного внешнего зависимого объекта его более легковесной версией. Идея заключается в том, что объект отражает реальное поведение, но не все. Fake object создается специально для тестирования. В качестве примеров можно рассмотреть эмулятор для конкретного приложения БД в памяти (fake database) или фальшивый веб-сервис.

## 7. Тестовое покрытие

**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.

Существуют следующие подходы 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).

**Тестовое покрытие на базе анализа потока управления** – оценка покрытия основанная на определении путей выполнения кода программного модуля и создания выполняемых тест кейсов для покрытия этих путей.

**Покрытие кода (Code Coverage)** – оценка покрытия исполняемого кода тестами, путем отслеживания непроверенных в процессе тестирования частей программного обеспечения. Различают несколько способов измерения покрытия кода. В Википедии представлены следующие определения:

- *Покрытие операторов* – каждая ли строка исходного кода была выполнена и протестирована?
- *Покрытие условий* – каждая ли точка решения (вычисления истинно ли или ложно выражение) была выполнена и протестирована?
- *Покрытие путей* – все ли возможные пути через заданную часть кода были выполнены и протестированы?
- *Покрытие функций* – каждая ли функция программы была выполнена?
- *Покрытие вход/выход* – все ли вызовы функций и возвраты из них были выполнены?
- *Покрытие комбинаций* – проверка всех возможных комбинаций результатов условий.

В глоссарии ISTQB предоставлен более полный список способов измерения покрытия кода. Однако общий смысл идентичен, но желающим все же рекомендую туда заглянуть.

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

$$T_{cov} = (L_{tc}/L_{code}) * 100\%, \text{ где:}$$

$T_{cov}$  – тестовое покрытие,  $L_{tc}$  – кол-ва строк кода, покрытых тестами,  $L_{code}$  – общее кол-во строк кода.

Покрытие кода является важной метрикой для обеспечения качества тестируемого приложения, особенно если речь о проектах со сложной логикой и большим объемом кода. Анализ покрытия кода выполняется с помощью специального инструментария, который позволяет проследить в какие строки, ветви и т.д. кода, были вхождения во время работы автотестов. Наиболее известные инструменты для проведения измерения покрытия кода: AQTime, Bounds Checker, Bullseye Coverage, Coverage Meter, Clover, NCover, IBM Rational PurifyPlus, Intel Compiler, Intel Code Coverage Tool Prototype, JetBrains. С помощью анализа покрытия кода можно оценить плотность покрытия автотестами исполняемого кода тестируемого приложения (можно ответить на вопрос «какой объем тестирования мы (наши автотесты) выполняем?»). При детальном анализе результатов покрытия кода автотестами можно оценить покрытие отдельных компонентов системы (т.е. можно ответить на вопросы: что и в каком объеме мы тестируем? в каких местах нужно оптимизировать покрытие?, какие места системы не проверяются тестами? и т.д.). Таким образом, зная данную метрику, станет ясно для каких тестовых случаев нужно создать новые тесты, или убрать дублирующие тесты. Данные мероприятия помогут увеличить значение метрики Code Coverage, что в свою очередь должно повысить качество кода и качество тестируемого приложения в целом. Естественно, чем выше показатель данной метрики – тем лучше, однако уже хорошо, если у вас покрыты тестами наиболее сложные и важные фрагменты кода.

*Проведение полного анализа покрытия кода вашего семестрового проекта не входит в задачу данной лабораторной работы. Поэтому мы ограничимся кратким обзором возможностей, предоставляемых Visual Studio.*

Анализ покрытия кода возможен при выполнении методов тестов с помощью обозревателя тестов. В таблице результатов отображается процент кода, который был выполнен в каждой сборке, классе и методе. Кроме того, редактор исходного кода показывает, какой код был протестирован (Рис. 9).

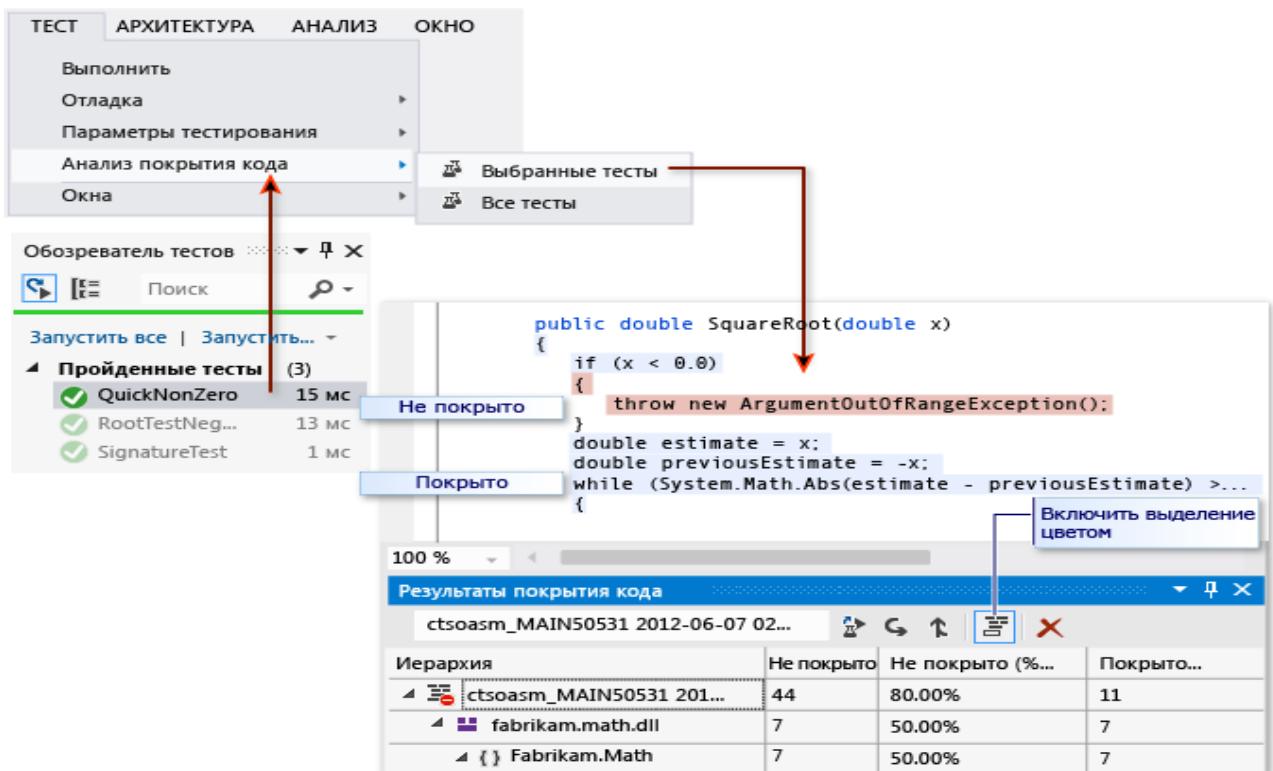


Рисунок 9. Вид окна обозревателя тестов

Анализ покрытия кода в модульных тестах в обозревателе тестов выполняется следующим образом.

- В меню Тест щелкните **Анализ покрытия кода**.
- Чтобы просмотреть, какие строки были выполнены, щелкните иконку для выделения цветом не выполнявшихся участков программы.
- Можно изменить цвета или использовать полужирный шрифт. Для этого последовательно щелкните Сервис, Параметры, Среда, Шрифты и цвета, Параметры для, Текстовый редактор. В разделе Отображаемые настройте элементы покрытия.

• Если результаты показывают низкое покрытие, проверьте, какие части кода не обрабатываются, и напишите несколько дополнительных тестов для их покрытия. Команды разработчиков обычно стремятся покрыть около 80 % кода. В некоторых случаях допустимо более низкое покрытие. Например, более низкое покрытие допустимо, когда некоторый код создается из стандартного шаблона.

Чтобы получить *точные результаты*, выполните следующие действия.

- Убедитесь, что оптимизация компилятора отключена.
- При работе с неуправляемым (машинным) кодом используйте отладочную сборку.
- Убедитесь, что PDB-файлы (файлы символов) создаются для каждой сборки.

Объем покрытого кода подсчитывается в *блоках*. Блок — это часть кода с одной точкой входа и точкой выхода. Если поток управления программы проходит через блок во время тестового запуска, то этот блок учитывается как покрытый. Количество раз, когда используется блок, не влияет на результат.

Результаты также можно отобразить в виде строк, щелкнув "Добавить" или удалить столбцы в заголовке таблицы. Если во время тестового запуска все блоки кода были обработаны в любой строке кода, она учитывается как одна строка. Если строка содержит

несколько блоков кода, одни из которых были обработаны, а другие — нет, то она учитывается как частичная строка.

Некоторые пользователи предпочитают считать в строках, поскольку процентные соотношения более точно соответствуют размеру фрагментов, которые можно увидеть в исходном коде. Длинный блок вычислений учитывается как единый блок, даже если он занимает большое количество строк.

В некоторых ситуациях будут использоваться разные блоки в коде в зависимости от данных теста. Поэтому может потребоваться объединить результаты различных тестовых запусков. *Например, при выполнении теста со входными данными "2" обнаружилось, что покрыто 50 % определенной функции. При выполнении теста во второй раз со входными данными "-2" в представлении расцветки покрытия видно, что покрыты другие 50% данной функции.* Теперь можно объединить результаты двух тестовых запусков, чтобы в отчете и представлении расцветки покрытия было видно, что покрыто 100 % данной функции.

Для этого щелкните иконку " **Объединить результаты покрытия кода**". Можно выбрать любые сочетания последних запусков или импортированных результатов. Если требуется объединить экспортованные результаты, в первую очередь необходимо их импортировать.

Щелкните **Экспортировать результаты покрытия кода**, чтобы сохранить результаты операции объединения.

## 8. Преимущества Unit-тестирования

### *Тесты как документация*

Юнит-тесты могут служить в качестве документации к коду. Грамотный набор тестов, который покрывает возможные способы использования, ограничения и потенциальные ошибки, ничуть не хуже специально написанных примеров, и, кроме того, его можно скомпилировать и убедиться в корректности реализации.

### *Упрощение архитектуры приложения*

Основное правило здесь формулируется следующим образом: «Реализуется только то, что действительно нужно». Если в teste описаны все сценарии и больше не удается придумать, как сломать логику, то нужно остановиться, привести код в более-менее приличный вид (выполнить рефакторинг) и со спокойной совестью лечь спать.

### *Возможность лучше разобраться в коде*

- Когда вы разбираетесь в плохо документированном сложном старом коде, попробуйте написать для него тесты. Это может быть непросто, но достаточно полезно, так как:

- они позволят вам убедиться, что вы правильно понимаете, как работает код;
- они будут служить документацией для тех, кто будет читать код после вас;
- если вы планируете рефакторинг, тесты помогут вам убедиться в корректности изменений.

## **Порядок выполнения лабораторной работы**

Разработать программу в соответствии со своим вариантом и подготовить к ней набор модульных тестов. Провести Unit-тестирование и анализ покрытия кода в модульных тестах.

### **Контрольные вопросы:**

1. Что за шаблон Arrange-Act-Assert?
2. Какие преимущества применения unit-тестирования?
3. Что такое тестовое покрытие?
4. Что такое моки и стабы?
5. Как осуществляется планирование тестов?
6. Какие правила организации тестов вы знаете?
7. Unit-тестирование относится с тестированию методом белого или черного ящика?
8. Как провести тестирование вашей программы методом белого ящика?

# ЛАБОРАТОРНАЯ РАБОТА № 7

## Экономические аспекты разработки ПО

**Цель работы:** Научиться проводить оценку стоимости и трудоемкости разработки ПО

**Продолжительность работы – 4 часа.**

### Содержание.

1. Линейный метод .....	115
2. Модель СОСМО .....	115
3. Методика функциональных точек .....	118
4. Порядок выполнения лабораторной работы.....	124
5. Вопросы .....	124

В любом программном проекте приходится балансировать между стоимостью, временем, качеством и объемом реализуемой функциональности.

Соответственно, расчет ресурсов, необходимых для реализации данного продукта с заданными требованиями к качеству, является одной из основных проблем в области управления проектами.

Первое, что необходимо понимать при оценке проекта, это то, что любая оценка это всегда *вероятностное утверждение*. Если мы просто скажем, что трудоемкость данного пакета работ составляет *M* *чел.\*мес.*, то это будет плохой оценкой потому, что одно единственное число ничего не скажет нам о вероятности того, что на реализацию этого пакета потребуется *не более, чем M* *чел.\*мес.* Вряд ли мы можем точно знать, что произойдет в будущем и сколько потребуется затрат на реализацию всего объема работ.

При расчете стоимости проекта возникают сложности учета огромного количества факторов, влияющих на жизненный цикл ПО. Как следствие, сегодня многие компании сталкиваются с серьезными проблемами в случае неправильных расчетов необходимых сроков:

при *недооценке* – непредвиденная траты дополнительных средств, недовольство заказчика невыполнением обязательств в срок, "бессонные ночи" сотрудников, сложность управления "обвальным" проектом, низкое качество конечного продукта, слаборазвитые и неполно соответствующие ТЗ функции системы и т. д;

при *переоценке* – отказ заказчика от контракта с данными условиями (как следствие, возможно, потеря рабочих мест) и т. д.

На современном рынке крупных программных систем потери могут исчисляться миллионами долларов. Точные оценки издержек производства программного обеспечения важны как разработчикам так и заказчику (клиенту). Они могут использоваться при переговорах о контракте, планировании, контроле и т. п. Таким образом, возникает реальная потребность в разработке методов и средств, позволяющих менеджеру оценить требуемые временные и человеческие ресурсы на основе всех имеющихся характеристик проекта: истории предыдущих подобных проектов, опыта и производительности сотрудников, специфики компании и т. п. Кроме того, требуется также возможность перерасчета и уточнения сроков и ресурсов уже на этапе разработки системы с учетом текущих тенденций, наблюдаемых при реализации проекта. Это поможет менеджеру своевременно обнаружить отклонения от установленного графика и принять соответствующие меры в управлении проектом.

Процесс оценки стоимости ПО является весьма сложной задачей. Он начинается на стадии анализа требований к программному обеспечению и продолжается на этапе реализации. Существует множество подходов к решению этой задачи, но до сих пор не

существует универсальной методики, дающей гарантированный результат. Можно только утверждать, что чем точнее сформулированы требования к разрабатываемому программному продукту, тем вернее можно оценить его стоимость. Для оценки стоимости разработки ПО чаще всего используются следующие методы:

- Линейный метод
- Методика COCOMO (Constructive Cost Model)
- Методика EFP IFPUG FPA (Early Function Points \ Function Point Analysis) – методика функциональных точек
  - Техника PERT (Project Evaluation and Review Technique)
  - Генетический подход
  - Метрики Холстеда

Последние три метода будут рассмотрены в рамках лекционного курса. Расчет трудоемкости вашего проекта будем проводить по методу функциональных точек.

## 1. Линейный метод

Этот довольно старый метод, несмотря на свою простоту (и даже примитивность), активно применяется по сей день.

Стоимость разработки определяется следующим образом:

$$C = T \times \Pi,$$

где  $C$  – стоимость,  $T$  – трудозатраты (например, в человеко-часах или человеко-месяцах), а  $\Pi$  – их удельная стоимость, которую определяют, в основном исходя из заработной платы и связанных с ней начислениях.

Трудозатраты вычисляют по следующей формуле:

$$T = P \times \Pi.$$

Здесь  $P$  – размер кода программы, чаще всего измеряется в строках (*LOC* – Lines Of Code),  $\Pi$  – временная производительность.

Недостаток такого подхода кроется в способе, которым измеряется результат – длина кода работающей программы существенно зависит от мастерства программиста, переиспользования кода, эффективности использования библиотек и других факторов. Кроме этого не учитывается жизненный цикл ПО.

## 2. Модель COCOMO.

Модель COCOMO оценивает трудоемкость разработки программного обеспечения (ПО) с помощью анализа функциональности программного кода только по «канкетным данным» проекта, т.е. по наиболее общим характеристикам, таким, как «масштаб проекта», «тип объекта проектирования», «пользователи объекта проектирования». Она позволяет оценить трудоемкость, сложность и стоимость будущего программного изделия еще на начальном этапе, когда о проекте еще почти ничего не известно.

Модель COCOMO состоит из трех последовательно уточняемых и детализируемых форм. Базовый уровень подходит для быстрых оценок разработки на ранней стадии проекта. Средний позволяет учесть факторы, определяемые при детализации проекта на стадии рабочего проекта. Детальный позволяет учитывать взаимовлияние отдельных фаз проекта друг на друга. COCOMO применим к трем классам проектов разработки ПО:

- Органический (Organic mode) – маленькие команды с хорошим опытом работы и не жесткими требованиями к разработке
- Полуразделенный вид (Intermediate/Semi-detached mode) – средние по размеру команды со смешанным опытом разработки и со смешанными требованиями (как жесткими, так и нет).

- Встроенный вид (Intered/Embedded mode) – разрабатываются с учетом множества жестких ограничений (по аппаратному, программному, операционному обеспечению и т.д.)

### **Базовый уровень**

Базовый уровень рассчитывает трудоемкость и стоимости разработки как функцию от размера программы. Размер выражается в оценочных тысячах строк кода (KLOC – *kilo lines of code*).

Вот базовые уравнения СОСМО:

$$\text{Трудоемкость} = a_b (KLOC)^{b_b} \text{ [человеко-месяцев]}$$

$$\text{Срок разработки или длительность} = c_b (\text{Трудоемкость})^{d_b} \text{ [месяцев]}$$

$$\text{Число разработчиков} = \text{Трудоемкость} / \text{Срок разработки} \text{ [человек]}$$

Коэффициенты  $a_b$ ,  $b_b$ ,  $c_b$  и  $d_b$  приведены в таблице 1.

*Таблица 1. Коэффициенты модели СОСМО Базового уровня*

Тип проекта	$a_b$	$b_b$	$c_b$	$d_b$
<b>Органический</b>	2.4	1.05	2.5	0.38
<b>Полуразделенный</b>	3.0	1.12	2.5	0.35
<b>Встроенный</b>	3.6	1.20	2.5	0.32

Базовый уровень СОСМО хорош для быстрой оценки стоимости разработки. Однако он не принимает во внимание различия в аппаратных ограничениях, качестве и опыте персонала, а также использованию современных техник и средств разработки и других факторов.

### **Средний уровень**

Средний уровень рассчитывает трудоемкость разработки как функцию от размера программы и множества «факторов стоимости», включающих субъективные оценки характеристик продукта, проекта, персонала и аппаратного обеспечения. Это расширение включает в себя множество из четырёх факторов, каждый из которых имеет несколько дочерних характеристик.

- Характеристики продукта
  - Требуемая надежность ПО
  - Размер БД приложения
  - Сложность продукта
- Характеристики аппаратного обеспечения
  - Ограничения быстродействия при выполнении программы
  - Ограничения памяти
  - Неустойчивость окружения виртуальной машины
  - Требуемое время восстановления
- Характеристики персонала
  - Аналитические способности
  - Способности к разработке ПО
  - Опыт разработки
  - Опыт использования виртуальных машин
  - Опыт разработки на языках программирования
- Характеристики проекта

- Использование инструментария разработки ПО
- Применение методов разработки ПО
- Требования соблюдения графика разработки

Каждому из этих 15 факторов ставится в соответствие рейтинг по шестибалльной шкале, начиная от «очень низкого» и до «экстра высокого» (по значению или важности фактора). Далее значения рейтинга заменяются множителями трудоемкости из нижеприведенной таблицы. Произведение всех множителей трудоемкости составляет Регулирующий фактор трудоемкости (РФТ). Обычно он принимает значения в диапазоне от 0.9 до 1.4. Коэффициенты представлены в таблице 2.

*Таблица 2. Коэффициенты рейтинга*

<b>Факторы стоимости</b>	Рейтинг					
	Очень низкий	Низкий	Средний	Высокий	Очень высокий	Критический
<b>Характеристики продукта</b>						
1. Требуемая надежность ПО	0.75	0.88	1.00	1.15	1.40	
2. Размер БД приложения		0.94	1.00	1.08	1.16	
3. Сложность продукта	0.70	0.85	1.00	1.15	1.30	1.65
<b>Характеристики аппаратного обеспечения***</b>						
4. Ограничения быстродействия при выполнении программы			1.00	1.11	1.30	1.66
5. Ограничения памяти			1.00	1.06	1.21	1.56
6. Неустойчивость окружения виртуальной машины		0.87	1.00	1.15	1.30	
7. Требуемое время восстановления		0.87	1.00	1.07	1.15	
<b>Характеристики персонала***</b>						
8. Аналитические способности	1.46	1.19	1.00	0.86	0.71	
9. Опыт разработки	1.29	1.13	1.00	0.91	0.82	
10. Способности к разработке ПО	1.42	1.17	1.00	0.86	0.70	
11. Опыт использования виртуальных машин	1.21	1.10	1.00	0.90		
12. Опыт разработки на языках программирования	1.14	1.07	1.00	0.95		
<b>Характеристики проекта***</b>						
13. Применение методов разработки ПО	1.24	1.10	1.00	0.91	0.82	
14. Использование инструментария разработки ПО	1.24	1.10	1.00	0.91	0.83	
15. Требования соблюдения графика разработки	1.23	1.08	1.00	1.04	1.10	

Формула модели СОСМО для среднего уровня принимает вид:

$$E = a_i (KLoC)^{b_i} PFT$$

где  $E$  – трудоемкость разработки ПО в человеко-месяцах,  $KLoC$  – оценочный размер программы в тысячах строках исходного кода, и  $PFT$  – регулирующий фактор, рассчитанный ранее. Коэффициенты  $a_i$  и показатель степени  $b_i$  представлены в следующей таблице.

Таблица 3. Коэффициенты Среднего уровня модели СОСМО

Тип проекта	$a_i$	$b_i$
Органический	3.2	1.05
Полуразделенный	3.0	1.12
Встроенный	2.8	1.20

Расчет времени разработки для среднего уровня СОСМО совпадает с расчетом для Базового уровня.

#### Детальный уровень

Детальный уровень включает в себя все характеристики среднего уровня с оценкой влияния данных характеристик на каждый этап процесса разработки ПО.

### 3. Методика функциональных точек EFP IFPUG FPA.

Анализ функциональных точек — стандартный метод измерения размера программного продукта с точки зрения пользователей системы. Метод разработан Аланом Альбрехтом (Alan Albrecht) в середине 70-х. Метод был впервые опубликован в 1979 году. В 1986 году была сформирована Международная Ассоциация Пользователей Функциональных Точек (International Function Point User Group — IFPUG), которая опубликовала несколько ревизий метода.

Метод предназначен для оценки на основе логической модели объема программного продукта количеством функционала, востребованного заказчиком и поставляемого разработчиком. Несомненным достоинством метода является то, что измерения не зависят от технологической платформы, на которой будет разрабатываться продукт, и он обеспечивает единообразный подход к оценке всех проектов в компании.

Оценка необходимых трудозатрат может быть выполнена на самых ранних стадиях работы над проектом и далее будет уточняться по ходу жизненного цикла, а явная связь между требованиями к создаваемой системе и получаемой оценкой позволяет заказчику понять, за что именно он платит, и во что выльется изменение первоначального задания.

При анализе методом функциональных точек надо выполнить следующую последовательность шагов (Рисунок 1):

1. Определение типа оценки.
2. Определение области оценки и границ продукта.
3. Подсчет функциональных точек, связанных с данными.
4. Подсчет функциональных точек, связанных с транзакциями.
5. Определение суммарного количества не выровненных функциональных точек (UFP).
6. Определение значения фактора выравнивания (FAV).
7. Расчет количества выровненных функциональных точек (AFP).

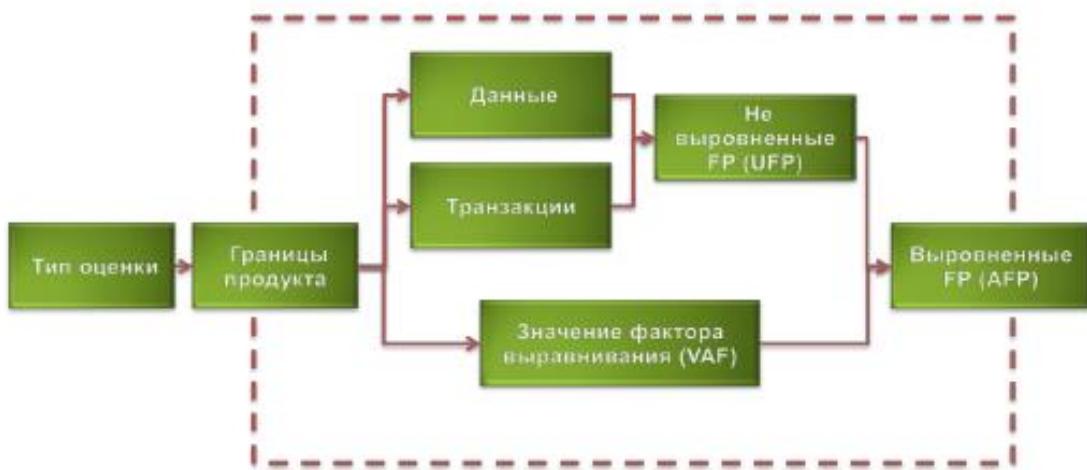


Рисунок 1. Процедура анализа по методу функциональных точек

**Первое**, что необходимо сделать, это определить тип выполняемой оценки. Метод предусматривает оценки трех типов:

1. *Проект разработки*. Оценивается количество функциональности поставляемой пользователям в первом релизе продукта.
2. *Проект развития*. Оценивается в функциональных точках проект доработки: добавление, изменение и удаление функционала.
3. *Продукт*. Оценивается объем уже существующего и установленного продукта.

**Второй шаг** — это определение области оценки и границ продукта. В зависимости от типа области оценки может включать:

- Все разрабатываемые функции (для проекта разработки)
- Все добавляемые, изменяемые и удаляемые функции (для проектов поддержки)
- Только функции, реально используемые, или все функции (при оценке продукта и/или продуктов).
- 

**Третий шаг**. Границы продукта (Рисунок 2) определяют:

- Что является «внешним» по отношению к оцениваемому продукту.
- Где располагается «граница системы», через которую проходят транзакции передаваемые или принимаемые продуктом, с точки зрения пользователя.
- Какие данные поддерживаются приложением, а какие — внешние.



Рисунок 2. Границы продукта в методе функциональных точек

К логическим данным системы относятся:

- Внутренние логические файлы (ILFs) — выделяемые пользователем логически связанные группы данных или блоки управляющей информации, которые поддерживаются внутри продукта.
- Внешние интерфейсные файлы (EIFs) — выделяемые пользователем логически связанные группы данных или блоки управляющей информации, на которые ссылается продукт, но которые поддерживаются вне продукта.

Примером логических данных (информационных объектов) могут служить: клиент, счет, тарифный план, услуга.

Кратко рассмотрим основные принципы метода. Общее количество функциональных точек программы зависит от количества элементарных процессов пяти типов (Рис. 3):

1. Входящие транзакции (External inputs (EI)) – транзакции, получающие данные от пользователя.
2. Исходящие транзакции (External outputs (EO)) – транзакции, передающие данные пользователю.
3. Взаимодействия с пользователем (External inquiries (EQ)) – интерактивные диалоги взаимодействия с пользователем (требующие от него каких-либо действий).
4. Файлы внутренней логики (Internal logical files) – файлы (логические группы информации), использующиеся во внутренних взаимодействиях системы.
5. Файлы внешних взаимодействий (External interface filese) – файлы, участвующие во внешних взаимодействиях с другими системами.

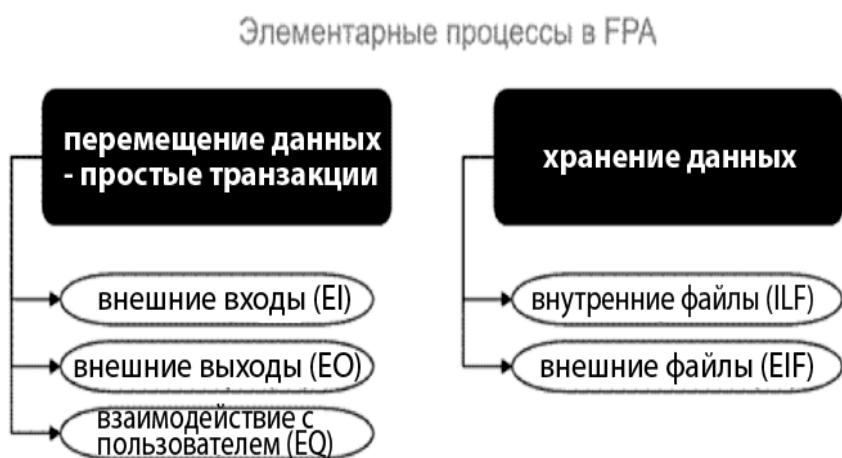


Рис. 3. Типы элементарных процессов используемых в методе FP.

В данной терминологии, транзакция – элементарный неделимый замкнутый процесс, представляющий значение для пользователя и переводящий продукт из одного консистентного состояния в другое.

Метод заключается в следующем:

Сначала выделяются функции разрабатываемого программного обеспечения, причем на уровне пользователей, а не программного кода.

Например, рассмотрим программный комплекс, реализующий различные методы сортировки одномерных массивов.

Одной из функций пользователя данного комплекса будет выбор метода, ее мы и будем описывать в качестве примера.

Следующим шагом метода будет подсчет количества факторов, приведенных ниже:

- внешние входы – различаются только те входы, которые по-разному влияют на функцию. Функция выбор метода имеет один внешний вход;
- внешние выходы – различными считаются выходы для различных алгоритмов. Представим, что наша функция выдает сообщение – текстовое описание выбранного метода, и вызывает другую функцию, непосредственно реализующую выбранный алгоритм сортировки, следовательно, она имеет два выхода;
- внешние запросы. В нашем примере таковых нет;
- внутренние логические файлы – группа данных, которая создается или поддерживается функцией, считается за единицу. В качестве внутреннего логического файла для нашей функции примем текстовый файл, содержащий описание алгоритмов;
- внешние логические файлы – пользовательские данные, находящиеся во внешних по отношению к данной функции файлах. Каждая группа данных принимается за единицу. Внешним по отношению к нашей функции является файл с результатом обработки.

Далее полученные значения умножаются на коэффициенты сложности для каждого фактора (по данным IFPUG) и суммируются для получения полного размера программного продукта. Значения этих коэффициентов приведены в таблице 2.

Параметр	Просто	Средне	Сложно
Внешние входы	3	4	6
Внешние выходы	4	5	7
Внешние запросы	3	4	6
Внутренние логические файлы	7	10	15
Внешние логические файлы	5	7	10

Каждому из этих типов назначают один из трех уровней сложности (1 = простой, 2 = средний, 3 = сложный), а каждой паре (*тип, уровень сложности*) ставят в соответствие вес представляющий собой количество не выровненных функциональных точек (UFP), который изменяется от 3 (для простой входящей транзакции) до 15 (для сложных внутренних файлов). Общая оценка размера в UFP рассчитывается как:

5 3

$$UFP = \sum_{i=1}^5 \sum_{j=1}^3 N_{ij} W_{ij},$$

где  $N_{ij}$  и  $W_{ij}$  – соответственно число и вес элементов системы класса  $i$  со сложностью  $j$ .

Например, если в системе – 2 простых входа ( $W_{ij} = 3$ ), 2 сложных выхода ( $W_{ij} = 7$ ) и 1 сложный внутренний файл ( $W_{ij} = 15$ ), то  $UFP = 2*3 + 2*7 + 1*15 = 35$ .

Это число функциональных точек может как непосредственно использоваться для оценки стоимости/трудоемкости так и может быть еще уточнено с помощью *фактора выравнивания (VAF)*, который вычисляется на основании характеристик общей сложности проекта, таких как степень распределенности обработки и хранения данных, требования к

производительности системы, требования к безопасности и т. д. При этом окончательная оценка размера в выровненных функциональных точках, рассчитывается как:

$$AFP = (UFP + CFP) - VAF,$$

где **CFP** – дополнительные функциональные точки, которые потребуются, например, для установки или миграции данных. Общая схема процедуры оценки представлена на рис. 2.

Параметр	Просто		Средне		Сложно	
	Количес тво	Коэффи циент	Количес тво	Коэффи циент	Количес тво	Коэффи циент
Внешние входы	1	3	0	4	0	6
Внешние выходы	1	4	1	5	0	7
Внешние запросы	0	3	0	4	0	6
Внутренние логические файлы	1	7	0	10	0	15
Внешние логические файлы	0	5	1	7	0	10

Для различных методы сортировки одномерных массивов

Размер нашей функции составит:

$$FP = 1 \times 3 + 1 \times 4 + 1 \times 5 + 1 \times 7 + 1 \times 7 = 26.$$

Это число является предварительной оценкой и нуждается в уточнении.

Следующим шагом в определении размера программного кода методом функциональных точек является присвоение веса (от 0 до 5) каждой *характеристике проекта*. Перечислим эти характеристики:

1. Требуется ли резервное копирование данных?
2. Требуется обмен данными?
3. Используются распределенные вычисления?
4. Важна ли производительность?
5. Программа выполняется на сильно загруженном оборудовании?
6. Требуется ли оперативный ввод данных?
7. Используется много форм для ввода данных?
8. Поля базы данных обновляются оперативно?
9. Ввод, вывод, запросы являются сложными?
10. Внутренние вычисления сложны?
11. Код предназначен для повторного использования?
12. Требуется преобразование данных и установка программы?
13. Требуется много установок в различных организациях?
14. Требуется поддерживать возможность настройки и простоту использования?

Значения для *данных характеристик* определяются следующим образом: 0 – никогда; 1 – иногда; 2 – редко; 3 – средне; 4 – часто; 5 – всегда.

Эти характеристики для примера функции сведены в табл. 9.3.

Определяется  $S$  – сумма всех весов.

И, наконец, уточненный функциональный размер вычисляется по формуле:

$$УФР = \PhiР \times (0,65 + 0,01 \times S). \quad (9.3)$$

Для различные методы сортировки одномерных массивов

Размер нашей функции составит:

$$\PhiР = 1 \times 3 + 1 \times 4 + 1 \times 5 + 1 \times 7 + 1 \times 7 = 26.$$

Это число является предварительной оценкой и нуждается в уточнении.

Уточненный функциональный размер функции выбор метода будет следующим:

$$УФР = 26 \times (0,65 + 0,01 \times 29) = 17,19.$$

Получившийся результат показывает, что функция выбор метода достаточно проста и не требует больших трудозатрат. Полученные значения затем используются для оценки стоимости проекта.

Значения для данных характеристик определяются следующим образом: 0 – никогда; 1 – иногда; 2 – редко; 3 – средне; 4 – часто; 5 – всегда.

Эти характеристики для примера функции сведены в табл. 9.3.

Характеристика	Значение в примере	Характеристика	Значение в примере
1	3	8	0
2	1	9	3
3	0	10	4
4	4	11	5
5	2	12	0
6	1	13	0
7	3	14	3

$$АFP = 26 \times (0,65 + 0,01 \times 29) = 17,19$$

$$LOC = a - UFP + b$$

Получившийся результат показывает, что функция выбор метода достаточно проста и не требует больших трудозатрат. Полученные значения затем используются для оценки стоимости проекта.

## Оценки количества SLOC на 1 UFP

Язык программирования	Среднее	Оптимистичная	Пессимистичная
Assembler	172	86	320
C	148	9	704
C++	60	29	178
C#	59	51	66
J2EE	61	50	100
JavaScript	56	44	65
PL/SQL	46	14	110
Visual Basic	50	14	276

«Function Point Programming Languages Table», Quantitative Software Management, Inc., 2005.

### Лабораторное задание.

1. Провести оценку размера ПО для вашего проекта в соответствии с методом функциональных точек.
3. Сравнить полученный результат с реальным размером вашего ПО

### Вопросы.

1. Перечислите известные вам способы оценки затрат на разработку ПО.
2. Когда целесообразно применение модели COCOMO?
3. Насколько точно, по вашему мнению, можно оценить стоимость разработки при использовании метода функциональных точек?

## ЛАБОРАТОРНАЯ РАБОТА №8

### Приемка программного обеспечения

**Цель работы:** Научиться методике проведения защиты проекта и приемо-сдаточных испытаний программного обеспечения – оценке результатов квалификационного тестирования программного обеспечения и системы в целом и документированию результатов оценки совместно с заказчиком, окончательную передачу программного обеспечения заказчику.

**Продолжительность работы – 4 часа.**

#### Содержание

1. Приемо-сдаточные испытания .....	125
2. Рекомендации по проведению презентаций .....	132
3. Порядок выполнения лабораторной работы.....	138
4. Вопросы.....	138

#### 1. Приемо-сдаточные испытания ПО

Важно понимать, что к моменту проведения приемо-сдаточных испытаний нельзя утверждать о готовности ПО к вводу в промышленную эксплуатацию, так как в результате этих испытаний наверняка будут выявлены дефекты или несоответствие продукта ТЗ, не обнаруженные Исполнителем. Приемо-сдаточные испытания проводятся с участием Заказчика и Исполнителя. Со стороны Заказчика обычно присутствуют представители подразделения эксплуатирующие и сопровождающие ПО. Необходимо учитывать, что разработанное ПО на момент испытаний, как правило, устанавливается непосредственно на оборудование и внутри информационной инфраструктуры Заказчика. А это может вызвать несовместимость с уже функционирующими программами. Все тонкости условий эксплуатации, безусловно, должны быть отражены в ТЗ, но предусмотреть все нюансы невозможно.

К приемо-сдаточным испытаниям должен быть подготовлен согласованный с Заказчиком набор документов, программный продукт, проверенный специалистами сопровождающего подразделения в рамках их должностных обязанностей, но без учета мнения конечных пользователей.

По завершению приемо-сдаточных испытаний составляется протокол, который отражает все недочеты в работе программного обеспечения и делается заключение о возможности ввода в опытную эксплуатацию или о необходимости проведения повторных испытаний после устранения недочетов.

Структура и оформление документа устанавливается в соответствии с ГОСТ 19.105-78. Документ «Программа и методика испытаний» должен содержать следующие разделы:

- объект испытаний;
- цель испытаний;
- требования к программе;
- требования к программной документации;
- средства и порядок испытаний;
- методы испытаний.

В зависимости от особенностей документа допускается вводить дополнительные разделы из п. 1.2 ГОСТ 19.301-79.

**Примечание:** ГОСТ Р ИСО/МЭК 12207-2010 более абстрактен в отношении приемо-сдаточных испытаний и сводит их к процессу: *6.4.8 Процесс поддержки приемки программных средств, включающему в себя цель, выходы, виды деятельности и задач.*

Мы, в рамках лабораторного практикума, будем придерживаться ГОСТ 19.105-78, как не противоречащему ГОСТ Р ИСО/МЭК 12207-2010, но более конкретному в формулировках и определению последовательности шагов при приемо-сдаточных испытаниях.

### **Объект испытаний**

В разделе «Объект испытаний» указывают наименование, область применения и обозначение испытуемой программы. Перечисленные сведения заимствуются из соответствующих разделов технического задания.

*Пример.*

#### Наименование

«Текстовый редактор для работы с файлами формата rtf».

#### Область применения

Программа предназначена к применению в профильных подразделениях на объектах заказчика.

#### Обозначение программы

Наименование темы разработки – «Разработка текстового редактора для работы с файлами формата rtf». Условное обозначение темы разработки (шифр темы) – «РТФ-007».

### **Цель испытаний**

В разделе «Цель испытаний» должна быть указана цель проведения испытаний – проверка соответствия характеристик разработанного ПО функциональным и отдельным иным видам требований, изложенным в документе Техническое задание.

Фактически, цель проведения испытаний – сдача работы заказчику. Формально – подтверждение соответствия функциональных и иных характеристик разработанной программы требованиям, сформулированным в техническом задании.

Основанием проведения испытаний является Приказ о проведении испытаний с составом приемочной комиссии.

*Пример:* Испытания проводятся на основании Приказа Директора ФГУП «Спецтяжмонтажсервисхозавтоматика» за № 128-б от 20 декабря 2017 г.

Приемосдаточные испытания должны проводиться на объекте заказчика в сроки...

Приемосдаточные испытания программы должны проводиться согласно разработанной (не позднее такого-то срока) исполнителем и согласованной с заказчиком Программы и методики испытаний.

Ход проведения приемо-сдаточных испытаний заказчик и исполнитель документируют в Протоколе испытаний. Испытания проводятся комиссией, в состав которой входят представители организаций заказчика и исполнителя. Состав комиссии утверждается Приказом.

Состав программной документации должен включать в себя:

1. техническое задание;
2. программу и методику испытаний;
3. руководство системного программиста;
4. руководство оператора;
5. ведомость эксплуатационных документов.

Испытания проводятся в два этапа:

- ознакомительный;
- испытания.

Перечень проверок, проводимых на 1-м этапе испытаний, должен включать в себя:

- проверку комплектности программной документации;
- проверку комплектности состава технических и программных средств.
- Методику проведения удобно вынести в приложение. Программа – отдельно, методика – отдельно.

- Перечень проверок, проводимых на 2-м этапе испытаний, должен включать в себя:

- проверку соответствия технических характеристик программы;
- проверку степени выполнения требований функционального назначения программы.

Методика проведения проверок, входящих в перечень по 2-му этапу испытаний, включает в себя количественные и качественные характеристики, подлежащие оценке. Качественные характеристики – все, что можно взвесить, измерить или просто сосчитать. Качественные – не требующие проведения измерений. Оцениваются экспертами.

*Пример.* При проверке «Текстового редактора для работы с файлами формата rtf» необходимо проверить возможность выполнения программой перечисленных ниже функций:

1. функции создания нового (пустого) файла.
2. функции открытия (загрузки) существующего файла.
3. функции редактирования открытого (далее – текущего) файла путем ввода, замены, удаления содержимого файла с применением стандартных устройств ввода.
4. функции редактирования текущего файла с применением буфера обмена операционной системы.
5. функции сохранения файла с исходным именем.
6. функции сохранения файла с именем, отличным от исходного.
7. функции отправки содержимого текущего файла электронной почтой с помощью внешней клиентской почтовой программы.
8. функции вывода оперативных справок в строковом формате (подсказок).
9. функции интерактивной справочной системы.
10. функции отображения названия программы, версии программы, копирайта и комментариев разработчика.

Приведенный выше перечень – результат вставки в настоящий документ п. «Требования к составу выполняемых функций» из технического задания.

В случае успешного проведения испытаний в полном объеме Исполнитель передает заказчику программное изделие, программную (эксплуатационную) документацию и т.д. Исполнитель совместно с заказчиком на основании Протокола испытаний утверждают Акт приемки-сдачи работ.

В случае выявления несоответствия разработанной программы отдельным требованиям ТЗ исполнитель проводит корректировку ПО и программной документации по результатам испытаний в сроки, согласованные с заказчиком.

По завершении корректировки программы и программной документации исполнитель и заказчик проводят повторные испытания согласно настоящей программы и методик в объеме, требуемом для проверки проведения корректировок.

Мелкие, несущественные недоработки могут быть устранены в рабочем порядке.

### **Требования к программе**

В разделе «Требования к программе» должны быть указаны требования, подлежащие проверке во время испытаний и заданные в техническом задании на программу. При проведении испытаний функциональные характеристики (возможности)

программы подлежат проверке на соответствие требованиям, изложенными в п. «Требования к составу выполняемых функций» технического задания.

Подлежат проверке требования, результат выполнения которых можно взвесить, измерить, посчитать. Выполнение многих требований бывает очевидно. Например требование «Программа должна обеспечивать свое выполнение под управлением операционной системы такой-то». Развернутый перечень требований предъявлять не обязательно, поскольку техническое задание входит в состав программных документов, предъявляемых для проведения испытаний. Но желательно.

### **Требования к программной документации**

В разделе «Требования к программной документации» должны быть указаны состав программной документации, предъявляемой на испытания, а также специальные требования, если они заданы в техническом задании.

Состав программной документации должен включать в себя:

- техническое задание;
- программу и методику испытаний;
- руководство системного программиста;
- руководство оператора;
- ведомость эксплуатационных документов.

### **Средства и порядок испытаний**

В разделе «Средства и порядок испытаний» должны быть указаны технические и программные средства, используемые во время испытаний, а также порядок проведения испытаний.

#### **Пример. Технические средства, используемые во время испытаний**

В состав технических средств должен входить IBM-совместимый персональный компьютер (ПЭВМ), включающий в себя:

- процессор Pentium-1000 с тактовой частотой, ГГц – 10, не менее;
- материнскую плату с FSB, ГГц – 5, не менее;
- оперативную память объемом, Тб – 10, не менее;
- и так далее...

Испытания проводятся на технических средствах, перечень заимствован из подраздела «Требования к составу и параметрам технических средств» ТЗ. Должна, очевидно, иметь место и разработанная программа.

Системные программные средства, используемые программой, должны быть представлены лицензионной локализованной версией операционной системы.

Для проведения испытаний предоставляется инсталляционная (установочная) версия разработанной программы.

Порядок проведения испытаний описан в п. «Перечень этапов испытаний». Испытания должны проводиться в нормальных климатических условиях по ГОСТ 22261-94. Пример условий проведения испытаний приведены ниже:

- температура окружающего воздуха, °C – 20 ± 5;
- относительная влажность, % – от 30 до 80;
- атмосферное давление, кПа – от 84 до 106;
- частота питающей электросети, Гц – 50 ± 0,5;
- напряжение питающей сети переменного тока, В – 220 ± 4,4.

Необходимым и достаточным условием завершения 1 этапа испытаний и начала 2 этапа испытаний является успешное завершение проверок, проводимых на 1 этапе (см. п. «Перечень проверок, проводимых на 1 этапе испытаний»).

Условием завершения 2 этапа испытаний является успешное завершение проверок, проводимых на 2 этапе испытаний.

Климатические условия эксплуатации, при которых должны обеспечиваться заданные характеристики, должны удовлетворять требованиям, предъявляемым к техническим средствам в части условий их эксплуатации.

При проведении испытаний заказчик должен обеспечить соблюдение требований безопасности, установленных ГОСТ 12.2.007.0–75, ГОСТ 12.2.007.3-75, «Правилами техники безопасности при эксплуатации электроустановок потребителей», и «Правилами технической эксплуатации электроустановок потребителей».

### **Порядок взаимодействия организаций, участвующих в испытаниях следующий.**

Исполнитель письменно извещает заказчика о готовности к проведению приемо-сдаточных испытаний не позднее чем за 14 дней до намеченного срока проведения испытаний.

Заказчик Приказом назначает срок проведения испытаний и приемочную комиссию, которая должна включать в свой состав представителей заказчика и исполнителя.

Заказчик письменно извещает сторонние организации, которые должны принять участие в приемо-сдаточных испытаниях.

Заказчик совместно с исполнителем проводят все подготовительные мероприятия для проведения испытаний на объекте заказчика, а так же проводят испытания в соответствии с настоящей программой и методиками.

Заказчик осуществляет контроль проведения испытаний, а также документирует ход проведения проверок в Протоколе проведения испытаний.

Персонал, проводящий испытания, должен быть аттестован на II квалификационную группу по электробезопасности (для работы с конторским оборудованием).

### **Методы испытаний**

В разделе «Методы испытаний» должны быть приведены описания используемых методов испытаний. Методы испытаний рекомендуется по отдельным показателям располагать в последовательности, в которой эти показатели расположены в разделах «Требования к программе» и «Требования к программной документации».

В методах испытаний должны быть приведены описания проверок с указанием результатов проведения испытаний (перечней тестовых примеров, контрольных распечаток тестовых примеров и т. п.). Сведения о методах проведения испытаний излагаются в документах Приложение А и Приложение Б.

### **Приложения**

В приложения могут быть включены тестовые примеры, контрольные распечатки тестовых примеров, таблицы, графики и т. п.

### **Приложение А (обязательное)**

- Методы проведения проверки комплектности программной документации.

Проверка комплектности программной документации на программное изделие производится визуально представителями заказчика. В ходе проверки сопоставляется состав и комплектность программной документации, представленной исполнителем, с перечнем программной документации.

Проверка считается завершенной в случае соответствия состава и комплектности программной документации, представленной исполнителем, перечню программной документации, приведенному в указанном выше пункте.

По результатам проведения проверки представитель заказчика вносит запись в Протокол испытаний.

**Пример:** «Комплектность программной документации соответствует (не соответствует) требованиям п. Перечень документов, предъявляемых на испытания настоящего документа». Протокол испытаний – п. 7 РД 50-34.698-90.

- Методы проведения проверки комплектности и состава технических и программных средств.

Проверка комплектности и состава технических и программных средств производится визуально представителем заказчика. В ходе проверки сопоставляется состав и комплектность технических и программных средств, представленных исполнителем, с перечнем технических и программных средств, приведенным в Табл X настоящего документа.

Комплектность системного блока, входящего в состав технических средств, может производиться по бланку заказа, если системный блок опечатан производителем или продавцом. Комплектность программных средств проводится также визуально.

*Пример: загрузилась операционная система, высветился логотип, версия – соответствует/не соответствует заявленной в техническом задании.*

Проверка считается завершенной в случае соответствия состава и комплектности технических и программных средств, представленных исполнителем, с перечнем технических и программных средств.

По результатам проведения проверки представитель заказчика вносит запись в Протокол испытаний – «Комплектность технических и программных средств соответствует (не соответствует) требованиям п. Технические средства, используемые во время испытаний настоящего документа».

## **Приложение Б (обязательное)**

Руководство оператора должно содержать подробные сведения о реализации всех функций программы. Чтобы не копировать указанные сведения в настоящий документ, достаточно ограничиться ссылками на подразделы Руководства оператора.

*Пример.* Для «Текстового редактора для работы с файлами формата rtf» может быть актуально следующее описание:

- Методы проверки выполнения функции создания нового (пустого) файла.

Проверка выполнения указанной функции выполняется согласно п. «Выполнение функции создания нового (безымянного) файла» руководства оператора.

*Проверка считается завершенной в случае соответствия состава и последовательности действий оператора при выполнении данной функции указанному выше подразделу руководства оператора. По результатам проведения проверки представитель заказчика вносит запись в Протокол испытаний – «п. такой-то выполнен».*

- Методы проверки выполнения функции открытия (загрузки) существующего файла.

Проверка выполнения указанной функции выполняется согласно п. «Выполнение функции открытия (загрузки) существующего файла» руководства оператора.

*Проверка считается завершенной в случае соответствия состава и последовательности действий оператора при выполнении данной функции указанному выше подразделу руководства оператора. По результатам проведения проверки представитель заказчика вносит запись в Протокол испытаний – «п. такой-то выполнен».*

- Методы проверки выполнения функции редактирования открытого (далее – текущего) файла путем ввода, замены, удаления содержимого файла с применением стандартных устройств ввода.

Проверка выполнения указанной функции выполняется согласно п. «Выполнение функции редактирования текущего файла путем ввода, замены, удаления содержимого файла с применением устройств ввода» руководства оператора.

Проверка считается завершенной в случае соответствия состава и последовательности действий оператора при выполнении данной функции указанному выше подразделу руководства оператора. По результатам проведения проверки представитель заказчика вносит запись в Протокол испытаний – «п. такой-то выполнен».

- Методы проверки выполнения функции редактирования текущего файла с применением буфера обмена операционной системы.

Проверка выполнения указанной функции выполняется согласно п. «Выполнение функции редактирования текущего файла с применением буфера обмена операционной системы» руководства оператора.

Проверка считается завершенной в случае соответствия состава и последовательности действий оператора при выполнении данной функции указанному выше подразделу руководства оператора. По результатам проведения проверки представитель заказчика вносит запись в Протокол испытаний – «п. такой-то выполнен».

- Методы проверки выполнения функции сохранения файла с исходным именем.

Проверка выполнения указанной функции выполняется согласно п. «Выполнение функции сохранения файла с исходным именем» руководства оператора.

Проверка считается завершенной в случае соответствия состава и последовательности действий оператора при выполнении данной функции указанному выше подразделу руководства оператора. По результатам проведения проверки представитель заказчика вносит запись в Протокол испытаний – «п. такой-то выполнен».

- Методы проверки выполнение функции сохранения файла с именем, отличным от исходного.

Проверка выполнения указанной функции выполняется согласно п. «Выполнение функции сохранения файла с именем, отличным от исходного» руководства оператора.

Проверка считается завершенной в случае соответствия состава и последовательности действий оператора при выполнении данной функции указанному выше подразделу руководства оператора. По результатам проведения проверки представитель заказчика вносит запись в Протокол испытаний – «п. такой-то выполнен».

- Методы проверки выполнения функции отправки содержимого текущего файла электронной почтой с помощью внешней клиентской почтовой программы

Проверка выполнения указанной функции выполняется согласно п. такому-то руководства оператора.

Проверка считается завершенной в случае соответствия состава и последовательности действий оператора при выполнении данной функции указанному выше подразделу руководства оператора. По результатам проведения проверки представитель заказчика вносит запись в Протокол испытаний – «п. такой-то выполнен».

- Методы проверки выполнения функции вывода оперативных справок в строковом формате (подсказок).

Проверка выполнения указанной функции выполняется согласно п. такому-то руководства оператора.

Проверка считается завершенной в случае соответствия состава и последовательности действий оператора при выполнении данной функции указанному выше подразделу руководства оператора. По результатам проведения проверки представитель заказчика вносит запись в Протокол испытаний – «п. такой-то выполнен».

- Методы проверки выполнения функции интерактивной справочной системы.

Проверка выполнения указанной функции выполняется согласно п. такому-то руководства оператора.

Проверка считается завершенной в случае соответствия состава и последовательности действий оператора при выполнении данной функции указанному выше подразделу руководства оператора. По результатам проведения проверки представитель заказчика вносит запись в Протокол испытаний – «п. такой-то выполнен».

- Методы проверки выполнение функции отображения названия программы, версии программы, копирайта и комментариев разработчика

Проверка выполнения указанной функции выполняется согласно п. такому-то руководства оператора.

Проверка считается завершенной в случае соответствия состава и последовательности действий оператора при выполнении данной функции указанному выше подразделу руководства оператора. По результатам проведения проверки представитель заказчика вносит запись в Протокол испытаний – «п. такой-то выполнен».

Вот, собственно, и вся Программа и методики испытаний. Программа и методика испытаний, разработанные согласно требований ГОСТ 19.301-79 – документ, достаточный (в целом) для проведения испытаний программных изделий. «Всеобъемлющим» же можно считать только программу и методику испытаний реализованными согласно п. 2.14. Руководящего документа РД 50-34.698-90. АВТОМАТИЗИРОВАННЫЕ СИСТЕМЫ. ТРЕБОВАНИЯ К СОДЕРЖАНИЮ ДОКУМЕНТОВ.

И хотя этот документ был написан еще в прошлом веке, актуальность его не потеряла значения. В случае приемо-сдаточных испытаний дорогостоящего ПО документацию по методике испытаний рекомендуется составлять в соответствии с этим четким документом.

## **2. Общие рекомендации по проведению презентаций.**

В рамках лабораторного практикума перед студентами не ставится задача полного выполнения *программы и методики испытаний разработанного в течение семестра ПО* – подготовка полного комплекта документов и выполнение всех процедур по приемке ПО задача непосильная для 4-х часовой работы.

Вместо этого каждая подгруппа студентов должна подготовить презентацию своего проекта и сделать доклад о разработанном ПО. Презентация к защите исследовательской работы или проекта предназначена для официального представления результатов проделанной работы и должна иметь четко определенную структуру. Впрочем, эта структура по логике подачи материала универсальна. Навык подготовки презентаций и выступления с докладом чрезвычайно важен. Даже не очень эффектный и убедительный проект можно представить так, что слушатели будут в восторге. И наоборот – неудачный доклад по очень интересной разработке может привести к ощущению абсолютной ненужности и плохо сделанной работы.

### **Самые популярные проблемы подачи информации:**

1. Непонятно о чем проект и какую пользу он несет.
2. Отсутствие адаптации презентации под аудиторию.
3. Цель презентации одна, наполнение презентации говорит совсем про другое.
4. Неумение выделить важное + акцент на непонятные или неинтересные детали.
5. Уверенность в том, что его продукт для всех и решает вообще все задачи в мире.
6. Грамматические ошибки.
7. и многое другое.

**Начнем с главного.** У вас должно быть: понимание *своего продукта*, анализ рынка и многое другое. Нужно четко осознавать, что вы создали, зачем и кому это может быть интересно.

**Определитесь с целью.** Необходимо четко понимать, какую цель ваша презентация преследует. Если это чисто научный доклад – акценты должны быть одни. Если цель продать продукт – акценты совсем другие.

**Кому вы будете презентовать ваш проект?** Есть огромная разница между группой специалистов и руководителями высоких рангов. Если в первом случае доклад должен содержать технические подробности вашего проекта, то во втором – краткую характеристику продукта, основные преимущества перед конкурентами, финансовый плюс вашей разработки и грандиозные перспективы развития.

Если вы уже делали доклад для специалистов и повторяете его для руководства, обязательно учитите все важные вопросы, которые были заданы. Если повторяете доклад – также не забудьте учсть все замечания!

**Начнем создание презентационного материала.** Напишите лично для себя, что вы хотите донести до слушателя. После окончания написания презентации прочитайте и оцените – соответствует ли она тому, что вы хотели доложить.

Презентация состоит из двух частей: демонстрация слайдов и сопровождение их тестом. Хотя выступление является единством слайдов и речи, все же презентация – это сопровождение доклада или выступления, а ни в коем случае не его замена. *Функция слайдов – поддержка выступления, а не наоборот.* Нарушение этого принципа приводит к весьма плачевным последствиям: докладчик вместо выступления просто зачитывает текст на слайдах. Таких ораторов слушатели не уважают, текст они могут и сами прочитать, если нужно.

Поэтому сначала необходимо разработать концепцию выступления, а затем уже браться за составление презентации. Для этого постарайтесь ответить себе на следующие вопросы:

1. Какова цель презентации?
2. Каковы особенности слушателей?
3. Обладают ли они достаточным уровнем знаний в предметной области, которой посвящен ваш доклад?
4. Будут ли другие участники с аналогичными докладами?
5. Сколько человек будет слушать ваш доклад?
6. Какова продолжительность презентации и планируемое содержание?
7. Как завладеть вниманием и сохранить интерес к выступлению?
8. Какие вопросы могут вам задать?
9. Какое оборудование вам требуется и нужны ли дополнительные демонстрационные материалы?

*Создайте план презентации.* Ниже приведено несколько схем организации содержания презентации зависящие собственно от темы доклада:

- Принцип решения проблемы;
- Принцип соответствия требованиям;
- Принцип пирамиды;
- Принцип анализа перспектив.

Рассмотрим основные акценты доклада в зависимости от используемых принципов.

#### **Принцип решения проблемы.**

*Вступление* – описание ключевой проблемы.

*Основная часть* – описание составляющих проблему компонентов и решение для этих компонентов.

*Заключение* – выводы о том, как все то, что вы доложили позволяет решить проблему в целом сейчас или в ближайшей перспективе.

### **Принцип соответствия требованиям.**

*Вступление* – описание требуемых характеристик продукта или услуги.

*Основная часть* – описание существующих характеристик продукта или услуги.

*Заключение* – выводы насколько продукт или услуга соответствуют требованиям.

### **Принцип пирамиды**

*Вступление* – описание ключевых пунктов.

*Основная часть* – детальное рассмотрение ключевых пунктов.

*Заключение* – подведение итогов по каждому ключевому пункту.

### **Принцип анализа перспектив**

*Вступление* – описание текущей ситуации.

*Основная часть* – изложение фактов, цифр, вариантов стратегий на будущее. Выбор стратегии.

*Заключение* – прогнозы на будущее исходя из выбранной стратегии.

### **Оптимальное количество слайдов для доклада на 10 минут – 13-15 слайдов.**

*Титульный лист. Он же вступление.* Напишите название проекта, а так же как вас зовут и кто вы (в рамках этого проекта или вообще). *Задайте себе вопрос, что должен или может подумать читатель увидев обложку вашей презентации?*

При создании основной части презентации старайтесь использовать структуры историй:

- Проблематика → Решение → Ваш результат;
- Факт → Вывод;
- История → Смысл → Развязка.

Обязательно надо обозначить какую проблему решает проект. В чем супер идея? *Создаете ли вы что-то уникальное?* Или вы комбинируете ранее используемые методы так, как никто не делал? Или ваши алгоритмы эффективнее?

Продемонстрируйте свой продукт и покажите, как он решает проблему. Для этого используйте снимки экрана (Screenshot), если ваш продукт не слишком сложен – продемонстрируйте работу демо-версии, если есть динамика в работе вашей программы – покажите анимацию.

Попробуйте оценить потенциальный рынок для вашей программы. Попробуйте ответить на вопрос кто ваши пользователи (целевая аудитория)? Сколько их? *Если вы не умеете определять целевую аудиторию и считать ее, не пишите "мужчины и женщины от 18 до 45 лет". Лучше ничего не писать.* Если сможете – покажите темп роста рынка.

Обязательно обозначьте существующие аналоги вашего продукта – вы, скорее всего, не уникальны. Ответьте на вопросы:

- Кто ваши конкуренты?
- Какие у нас отличия? Что упустили конкуренты?
- В чем вы лучше?

Если это возможно – попробуйте оценить ваш проект с точки зрения бизнеса. Покажите барьеры входа для конкурентов:

- Сложно ли повторить ваш проект/технологию?

- Есть ли патенты? *Если патент нужен или возможен, лучше не упускать из виду этот пункт.*
- Использованы ли уникальные технологии? *Если да, обязательно выделите этот пункт и обоснуйте почему.*
- Нужны ли уникальные специалисты? *Если да, обязательно выделите этот пункт и обоснуйте почему.*

*В заключительной части укажите фазу развития проекта и, если проект не завершен, график релизов.*

*Не последнее место в том, как будет воспринят ваш доклад, занимает стиль оформления презентации.*

Презентация предполагает сочетание информации различных типов: текста, графических изображений, музыкальных и звуковых эффектов, анимации и видеофрагментов. Поэтому необходимо учитывать специфику комбинирования фрагментов информации различных типов. Кроме того, оформление и демонстрация каждого из перечисленных типов информации также подчиняется определенным правилам. Для текстовой информации важен выбор шрифта, для графической — яркость, контрастность и насыщенность цвета. Для наилучшего их совместного восприятия необходимо оптимальное взаиморасположение на слайде.

Многие дизайнеры утверждают, что *законов и правил в дизайне нет*. Есть советы, рекомендации, приемы. Дизайн, как всякий вид творчества, искусства, как всякий способ одних людей общаться с другими, как язык, как мысль — обойдет любые правила и законы. Однако можно привести определенные рекомендации, которые следует соблюдать, во всяком случае до тех пор, пока вы не почувствуете в себе силу и уверенность сочинять собственные правила и рекомендации.

При разработке презентации важно учитывать, что материал на слайде можно разделить на главный и дополнительный. Главный материал необходимо выделить, чтобы при демонстрации слайда он нёс основную смысловую нагрузку. Выделить можно размером текста или объекта, цветом, спецэффектами, порядком появления на экране. Дополнительный материал предназначен для подчёркивания основной мысли слайда.

Старайтесь избегать использования слайда «картинка, обтекаемая текстом». Иллюстрацию лучше разместить на отдельном слайде, подписав под ней основную информацию. Или, как вариант, расположить текст снизу или сбоку от картинки. Текст в этом случае лучше воспринимается.

Вставляемые фотографии или картинки должны быть хорошего качества и достаточно большого размера, иначе при растягивании они теряют резкость, чем могут только испортить эффект от их использования. Допускается использовать

Графическая информация – рисунки, фотографии, диаграммы – призваны дополнить текстовую информацию или передать ее в более наглядном виде. Поэтому желательно избегать в презентации рисунков, не несущих смысловой нагрузки, если они не являются частью стилевого оформления. Цвет графических изображений не должен резко контрастировать с общим стилевым оформлением слайда.

Если графическое изображение используется в качестве фона, то текст на этом фоне должен быть хорошо читаем.

*Не рекомендуется использовать в стилевом оформлении презентации более 3 цветов и более 3 типов шрифта, все слайды презентации должны быть выдержаны в одном стиле.* Информационных блоков не должно быть слишком много (3-6), рекомендуемый размер одного информационного блока — не более 1/2 размера слайда. Желательно присутствие на странице блоков с разнотипной информацией (текст, графики,

диаграммы, таблицы, рисунки), дополняющей друг друга, ключевые слова в информационном блоке необходимо выделить.

Мелких (менее 1/5 экрана) картинок не должно быть вообще.

Информационные блоки лучше располагать горизонтально. Связанные по смыслу блоки – слева направо, наиболее важную информацию следует поместить в центр слайда. Логика предъявления информации на слайдах и в презентации должна соответствовать логике ее изложения. Помимо правильного расположения текстовых блоков, нужно не забывать и о самом тексте. В нем ни в коем случае не должно содержаться орфографических ошибок!

В таблицах не рекомендуется делать более 4 строк и 4 столбцов. Наиболее значимую информацию в таблице рекомендуется выделять либо цветом, либо жирным шрифтом. Диаграммы не должны включать более 5 элементов.

Если требуются диаграммы и таблицы с большим объемом данных – подготовьте раздаточный материал.

#### *Правила шрифтового оформления:*

1. Для разных видов объектов рекомендуются разные размеры шрифта. Заголовок слайда лучше писать размером шрифта 22-28, подзаголовок и подписи данных в диаграммах – 20-24, текст, подписи и заголовки осей в диаграммах, информацию в таблицах – 18-22.
2. Для выделения заголовка используйте полужирный шрифт, ключевых слов – полужирный или подчёркнутый шрифт. Не злоупотребляйте подчеркиванием! Для оформления второстепенной информации и комментариев – курсив.
3. Чтобы повысить эффективность восприятия материала слушателями, помните о «принципе шести»: в строке – шесть слов, в слайде – шесть строк.
4. ***Используйте шрифт одного названия на всех слайдах презентации!***
5. Для хорошей читаемости презентации с любого расстояния в зале текст лучше набирать понятным шрифтом. Это могут быть шрифты Arial, Bookman Old Style, Calibri, Tahoma, Times New Roman, Verdana.
6. Точка в конце заголовка и подзаголовках, выключенных отдельной строкой, не ставится. Если заголовок состоит из нескольких предложений, то точка не ставится после последнего из них. Порядковый номер всех видов заголовков, набираемый в одной строке с текстом, должен быть отделен пробелом независимо от того, есть ли после номера точка.
7. Точка не ставится в конце подрисунковой подписи, в заголовке таблицы и внутри нее. При отделении десятичных долей от целых чисел лучше ставить запятую (0,158), а не точку (0.158).
8. Не выносите на слайд излишне много текстового материала. Из-за этого восприятие слушателей перегружается, нарушая концентрацию внимания.
9. Шрифты с засечками читаются легче, чем гротески (шрифты без засечек);
10. Для основного текста не рекомендуется использовать прописные буквы.
11. Шрифтовой контраст можно создать посредством:
  - размера шрифта,
  - толщины шрифта,
  - начертания,
  - формы,
  - направления и цвета.

***Если у вас есть возможность заменить текст – картинкой, таблицей, графиком, фотографией – замените его.***

#### *Правила выбора цветовой гаммы.*

1. Цветовая гамма должна состоять не более чем из двух – трех цветов. Рекомендации по сочетаемости цветов легко найти в интернете.

2. Существуют не сочетаемые комбинации цветов и их лучше не использовать, если только вы не хотите специально шокировать слушателей.
3. Черный цвет имеет негативный (мрачный) подтекст.
4. Белый текст на черном фоне читается плохо.
5. Помните, что черный и синий цвета воспринимаются лучше всего.
6. Красный цвет агрессивный, но он концентрирует внимание к элементам.

#### *Правила общей композиции.*

1. На полосе не должно быть больше семи значимых объектов. Психологи полагают, что человек не в состоянии запомнить за один раз более семи пунктов чего-либо.
2. Логотип на полосе должен располагаться справа внизу (слева наверху и т. д.).
3. Дизайн должен быть простым, а текст — коротким.
4. Изображения домашних животных, детей, женщин и т.д. являются положительными образами.
5. Крупные объекты в составе любой композиции смотрятся довольно неважно.

*Не стоит забывать, что на каждое подобное утверждение есть сотни примеров, доказывающих обратное. Поэтому приведенные утверждения нельзя назвать общими и универсальными правилами дизайна, они верны лишь в определенных случаях.*

#### *Анимация*

Анимационные эффекты используются для привлечения внимания слушателей или для демонстрации динамики развития какого-либо процесса. В этих случаях использование анимации оправдано, но не стоит чрезмерно насыщать презентацию такими эффектами, иначе это вызовет негативную реакцию аудитории. Анимация должна использоваться по минимуму и лишь тогда, когда на ней лежит функциональная нагрузка.

С помощью анимации хорошо выделять ключевые слова, цифры, обозначать выводы. Будет лучше, если анимация настроена на выделение цветом, а не на движение букв или объектов на экране.

#### *Звук*

Звуковое сопровождение должно отражать суть или подчеркивать особенность темы слайда, презентации. Если это фоновая музыка, то она должна не отвлекать внимание слушателей и не заглушать слова докладчика.

После создания презентации и ее оформления, необходимо отрепетировать ее показ и свое выступление, проверить, как будет выглядеть презентация в целом (на экране компьютера или проекционном экране), насколько скоро и адекватно она воспринимается из разных мест аудитории, при разном освещении, шумовом сопровождении, в обстановке, максимально приближенной к реальным условиям выступления. Имейте в виду: как правило, репетиция короче реального доклада на 20%!

#### **Основные правила выступления**

Стройте выступление на аргументах, а не на слайдах. Докладчик должен вести аудиторию не от слайда к слайду, а от тезиса к аргументу, от аргумента к примеру, от вывода к выводу.

Не стоит говорить "перейдем на слайд 7". Лучше будет сказать: "как именно мы решаем эту проблему, показано на слайде 7". Нельзя говорить "посмотрите на следующий слайд". Правильнее будет сказать, например: "Из информации, показанной на слайде следует..."

При распределении времени доклада, нужно знать, что 1-2 минуты нужно отвести на *вступление*, 6-7 на *основную часть*, 1-2 минуты на *заключение*.

Две первые и две последние фразы запоминаются из выступления лучше всего. Позаботьтесь о том, чтобы они как-то проявлялись на начальных и конечных слайдах.

Лучше всего зрительно запоминаются образы, символы, картинки, расположенные в левом верхнем и правом нижнем углу. Проследите за этим на самых значимых слайдах.

**Важно!** Готовьтесь к выступлению! Выступление должно быть подготовлено, прорепетировано и даже отхронометрировано. Большинство выступающих этим простым правилом пренебрегает, а вот аудитория замечает плохую подготовку сразу. Мало кто умеет компенсировать недостаточность предварительной подготовки импровизацией! Не стесняйтесь в процессе доклада смотреть на часы – вряд ли у вас есть точные часы в голове. Чаще всего неопытный докладчик либо затягивает доклад (что плохо) либо оттарабанивает текст за 3-4 минуты (что просто ужасно!).

**Помните**, что аудитория – это живые люди. Понимание и принятие вашей точки зрения достигается *не на уровне* "правильно – неправильно", а на *уровне* "согласен – не согласен". За те средние 20 минут, что отводятся на презентацию, докладчик должен заставить слушателей поверить ему. Если он не смог понравиться, аудитории, то потратил зря и эти 20 минут, и все время, которое выполнялся проект и готовилось выступление.

**Верьте в то, что говорите!** Как бы складно ни была написана ваша речь – она не тронет никого, если вы просто прочтете ее. Ключик к сердцам очень прост: дайте аудитории почувствовать в вас человека, почувствовать, что тема вам близка и интересна.

Если вы сами не верите в то, что говорите – вам никто не поверит. Хотя бы на время выступления, но вы должны верить в то, что тема вашего доклада необыкновенно важна, необыкновенно интересна и вся работа сделана на высочайшем уровне.

**Позволяйте себе эмоции!** Не используйте жаргонные слова, "мусорные" слова, не делайте пауз в докладе. Сконцентрироваться на 10-15 минут может даже неподготовленный человек.

*Удачи! И помните: удача любит хорошо подготовленных людей!*

### **Лабораторное задание**

1. Подготовьте перечень документов, необходимых для проведения приемо-сдаточных испытаний по вашему проекту.
2. Подготовьте презентацию и проведите доклад о разработанной вами программе.

### **Вопросы**

1. Готово ли ПО к вводу в промышленную эксплуатацию к моменту проведения приемо-сдаточных испытаний? Почему?
2. Что должен содержать документ «Программа и методика испытаний»?
3. Что должен включать в себя состав программной документации?
4. Порядок взаимодействия организаций, участвующих в испытаниях.