Министерство образования Республики Беларусь
Учреждение образования
«Гомельский государственный технический университет
имени П.О.Сухого»

Факультет автоматизированных и информационных систем

Кафедра «Информационные технологии»

ЛАБОРАТОРНЫЙ ПРАКТИКУМ
по дисциплине
«ПРОГРАММНАЯ ИНЖЕНЕРИЯ»
для студентов специальности
1-40 05 01 «Информационные системы и технологии (по направлениям)»
направление специальности 1-40 05 01-01 «Информационные системы и технологии (в проектировании и производстве)»

Трубенок Д.Н.

Гомель 2018

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Дисциплина «Программная инженерия» позволяет изучить современные инженерные принципы создания надежного, качественного программного обеспечения, удовлетворяющего предъявленным требованиям.

Термин «программная инженерия» появился впервые в 1968 году на конференции НАТО и предназначался для провоцирования поиска решений, происходившего в то время «кризиса программного обеспечения». С тех пор это переросло в профессию программного инженера (англ. software engineer) и область исследований, посвящённых созданию программного обеспечения, более качественного, доступного, лучше поддерживаемого и быстрее разрабатываемого.

Программная инженерия является разделом информатики и связана с менеджментом. Она также считается частью общей системной инженерии.

Курс направлен на изучение терминологии и принципов программной инженерии, а также на формирование у студентов понимания необходимости применения данных принципов программной инженерии.

Лабораторный практикум нацелен на обучение студентов работе в команде и на изучение жизненного цикла программного обеспечения от стадии проектирования до стадии выпуска в производство.

ЛАБОРАТОРНАЯ РАБОТА №1

«РАЗРАБОТКА ГРУППЫ ПЛАНОВ ОБЕСПЕЧЕНИЯ ЖИЗНЕННОГО ЦИКЛА И РАСПРЕДЕЛЕНИЯ РЕСУРСОВ ПРОЕКТА СЛОЖНОГО КОМПЛЕКСА ПРОГРАММ»

Цель работы: изучить разработку группы планов обеспечения жизненного цикла и распределения ресурсов проекта сложного комплекса программ.

ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Техническое задание – исходный документ на проектирование технического объекта (изделия). ТЗ устанавливает основное назначение разрабатываемого объекта, его технические характеристики, показатели качества и технико-экономические требования, предписание по выполнению необходимых стадий создания документации (конструкторской, технологической, программной и т. д.) и её состав, а также специальные требования. Техническое задание является юридическим документом – как приложение включается в договор между заказчиком и исполнителем на проведение проектных работ и является его основой: определяет порядок и условия работ, в том числе цель, задачи, принципы, ожидаемые результаты и сроки выполнения. То есть должны быть объективные критерии, по которым можно определить, сделан ли тот или иной пункт работ или нет. Все изменения, дополнения и уточнения формулировок ТЗ обязательно согласуются с заказчиком и им утверждаются. Это необходимо и потому, что в случае обнаружения в процессе решения проектной задачи неточностей или ошибочности исходных данных возникает необходимость определения степени вины каждой из сторон-участниц разработки, распределения понесенных в связи с этим убытков. Техническое задание, как термин в области информационных технологий – это юридически значимый документ, содержащий исчерпывающую информацию, необходимую для постановки задач исполнителям на разработку, внедрение или интеграцию программного продукта, информационной системы, сайта, портала либо прочего ИТ сервиса [1-4].

Структура технического задания:

1. Общие положения
2. Название и цели создания системы
3. Характеристика объекта автоматизации
4. Требования к системе
5. Состав и содержание работ по созданию системы
6. Порядок контроля и приемки системы (дедлайны)
7. Требования к созданию и содержанию работ по подготовке объекта автоматизации к вводу системы в действие
8. Требования к документированию
9. Источники разработки

Техническое задание позволяет:

– исполнителю – понять суть задачи, показать заказчику «технический облик» будущего изделия, программного изделия или автоматизированной системы;

– заказчику – осознать, что именно ему нужно;

– обеим сторонам – представить готовый продукт;

– исполнителю – спланировать выполнение проекта и работать по намеченному плану;

– заказчику – требовать от исполнителя соответствия продукта всем условиям, оговорённым в ТЗ;

– исполнителю – отказаться от выполнения работ, не указанных в ТЗ;

– заказчику и исполнителю – выполнить попунктную проверку готового продукта (приёмочное тестирование – проведение испытаний);

– избежать ошибок, связанных с изменением требований (на всех стадиях и этапах создания, за исключением испытаний).

В зависимости от ожиданий заказчика существует три альтернативы для выбора шаблона Технического задания. Если заказчик требует оформления документации в соответствии с государственным стандартом, выбор делается в сторону стандарта ГОСТ 34.602-89. Подготовка Технического задания по ГОСТ 34.602-89 требует значительных временных затрат.

Если поставлены сжатые сроки подготовки ТЗ и заказчик не требует оформления документации в соответствии с государственным стандартом, то можно использовать шаблон технического задания по стандарту IEEE Std 830. Стандарт IEEE Std 830 предполагает, что детальные требования могут быть обширными и не существует оптимальной структуры для всех систем. По этой причине, стандартом рекомендуется обеспечивать такое структурирование детальных требований, которое делает их оптимальными для понимания. Стандартом рекомендуются различные способы структурирования детальных требований для различных классов систем.

Существует и третья альтернатива для выбора шаблона Технического задания, когда заказчик предлагает использовать принятый в компании Корпоративный шаблон для описания требований к информационным системам.

Задание.

Разбиться на группы по два (допустимо три) человека. Выбрать задание из таблицы 1 (Все задания связаны с разработкой компьютерной игры. Разрешается предложить свой вариант с предварительным согласованием с преподавателем). На основе задания сформулировать техническое задание, которое будет включать: этапы разработки, необходимый функционал, разделение ресурсов проекта (распределение задач для каждого члена команды), сроки выполнения этапов.

Проекты можно разрабатывать на *C#* или *Java* (на выбор группы).

Таблица 1 – Варианты заданий

Вариант	**Задание**
1	Змейка
2	Тетрис
3	Крестики-нолики
4	Пакман
5	Гоночки
6	Танчики
7	Pocket tanks
8	Worms
9	Морской бой
10	Bomberman
11	Теннис 2D
12	Воллейбол 2D
13	Шашки
14	Косынка
15	Свинка (Карточная игра. Выигрывает тот, кто откроет больше парных карт)
16	Астероиды
17	Fluffy Bird

Контрольные вопросы

1. Что такое техническое задание.
2. Структура технического задания.
3. Какие есть шаблоны для технического задания.

ЛАБОРАТОРНАЯ РАБОТА №2

«РАБОТА С СИСТЕМОЙ КОНТРОЛЯ ВЕРСИЙ»

Цель работы: изучить работу с системой контроля версий *git*.

ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Система управления версиями (*Version Control System*, *VCS*) – 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*).

Виды *VCS*:

– локальные системы контроля версий;

– централизованные системы контроля версий;

– децентрализованные системы контроля версий.

Локальные системы контроля версий

Многие люди в качестве метода контроля версий применяют копирование файлов в отдельную директорию (возможно даже, директорию с отметкой по времени, если они достаточно сообразительны). Данный подход очень распространён из-за его простоты, однако он невероятно сильно подвержен появлению ошибок. Можно легко забыть, в какой директории вы находитесь, и случайно изменить не тот файл или скопировать не те файлы, которые вы хотели.

Для того, чтобы решить эту проблему, программисты давным-давно разработали локальные СКВ с простой базой данных, которая хранит записи о всех изменениях в файлах, осуществляя тем самым контроль ревизий.

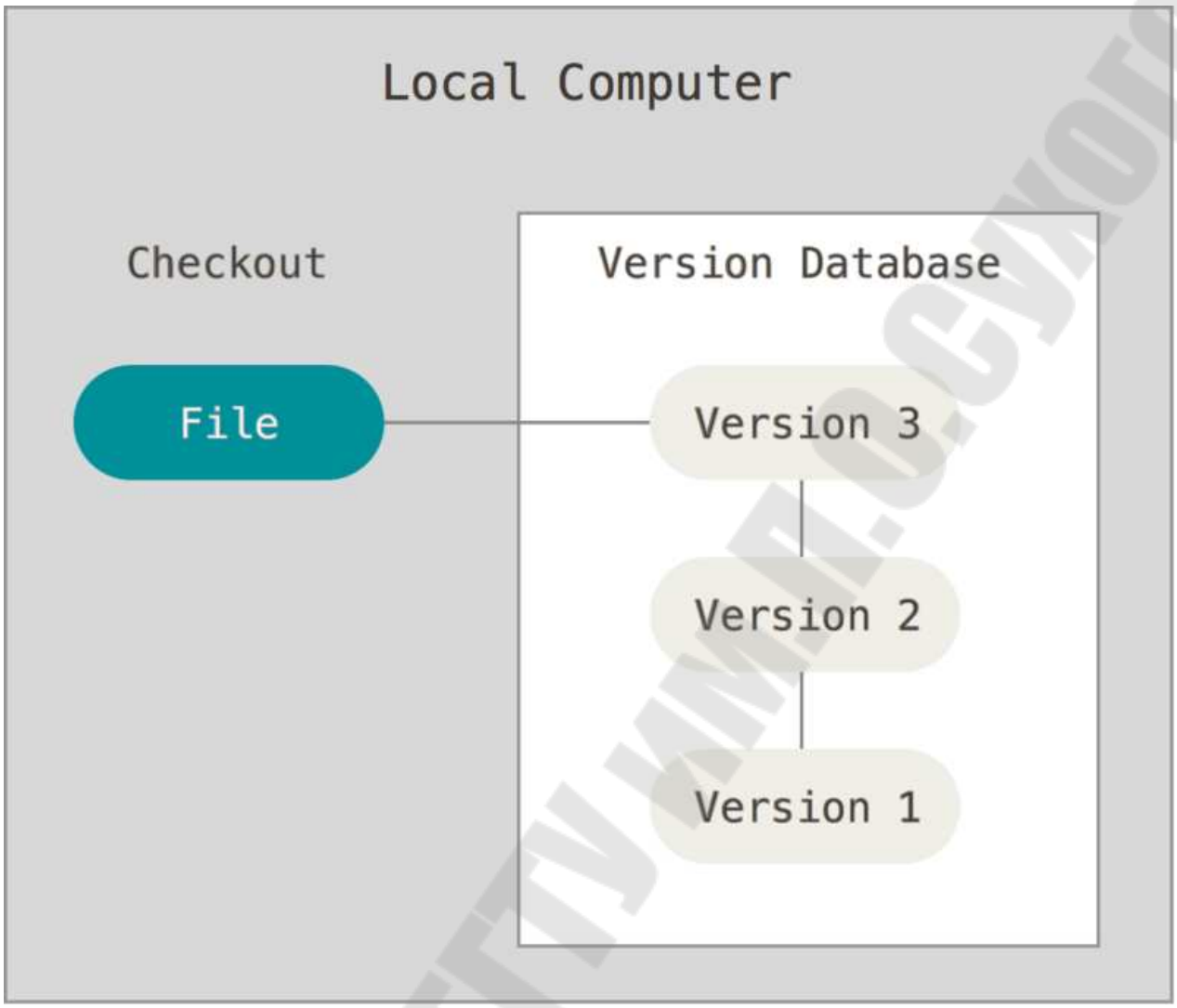

Рисунок 2.1 – Локальный контроль версий

Одной из популярных СКВ была система *RCS*, которая и сегодня распространяется со многими компьютерами. Даже популярная операционная система Mac OS X предоставляет команду rcs, после установки *Developer Tools*. *RCS* хранит на диске наборы патчей (различий между файлами) в специальном формате, применяя которые она может воссоздавать состояние каждого файла в заданный момент времени.

Централизованные системы контроля версий

Следующая серьёзная проблема, с которой сталкиваются люди, – это необходимость взаимодействовать с другими разработчиками. Для того, чтобы разобраться с ней, были разработаны централизованные системы контроля версий (ЦСКВ). Такие системы, как: CVS, *Subversion* и *Perforce*, имеют единственный сервер, содержащий все версии файлов, и некоторое количество клиентов, которые получают файлы из этого централизованного хранилища. Применение ЦСКВ являлось стандартом на протяжении многих лет.

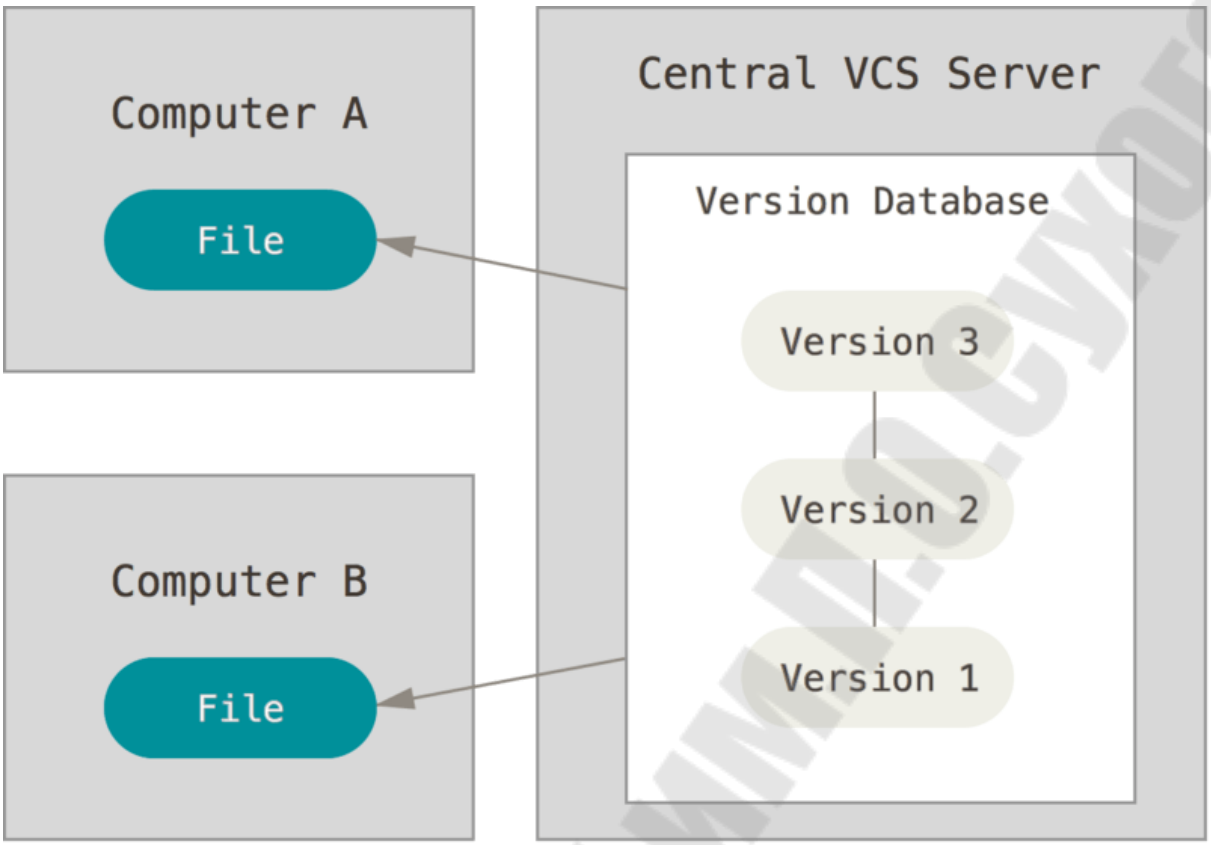

Рисунок 2.2 – Централизованный контроль версий

Такой подход имеет множество преимуществ, особенно перед локальными СКВ. Например, все разработчики проекта в определённой степени знают, чем занимается каждый из них. Администраторы имеют полный контроль над тем, кто и что может делать, и гораздо проще администрировать ЦСКВ, чем оперировать локальными базами данных на каждом клиенте.

Несмотря на это, данный подход тоже имеет серьёзные минусы. Самый очевидный минус – это единая точка отказа, представленная централизованным сервером. Если этот сервер выйдет из строя на час, то в течение этого времени никто не сможет использовать контроль версий для сохранения изменений, над которыми он работает, а также никто не сможет обмениваться этими изменениями с другими разработчиками. Если жёсткий диск, на котором хранится центральная БД, повреждён, а своевременные бэкапы отсутствуют, вы потеряете всё – всю историю проекта, не считая единичных снимков репозитория, которые сохранились на локальных машинах разработчиков. Локальные СКВ страдают от той же самой проблемы – когда вся история проекта хранится в одном месте, вы рискуете потерять всё.

Децентрализованные системы контроля версий

Здесь в игру вступают децентрализованные системы контроля версий (ДСКВ). В ДСКВ (таких как *Git*, *Mercurial*, *Bazaar* или *Darcs*), клиенты не просто скачивают снимок всех файлов (состояние файлов на определённый момент времени): они полностью копируют репозиторий. В этом случае, если один из серверов, через который разработчики обменивались данными, умрёт,

любой клиентский репозиторий может быть скопирован на другой сервер для продолжения работы. Каждая копия репозитория является полным бэкапом всех данных.

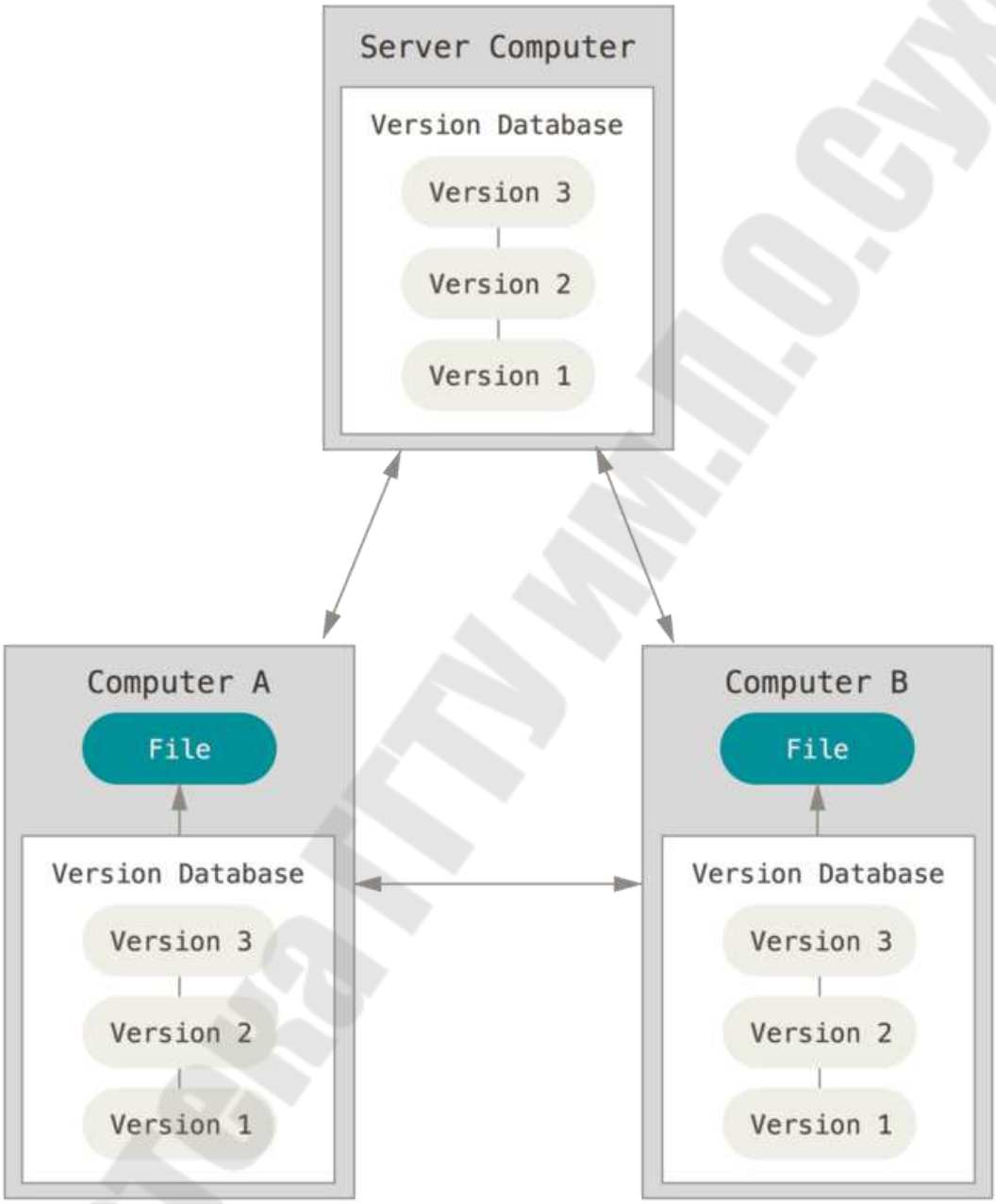

Рисунок 2.3 – Децентрализованный контроль версий

Более того, многие ДСКВ могут одновременно взаимодействовать с несколькими удалёнными репозиториями, благодаря этому вы можете работать с различными группами людей, применяя различные подходы единовременно, в рамках одного проекта. Это позволяет применять сразу несколько подходов в разработке, например, иерархические модели, что совершенно невозможно в централизованных системах.

Задание.

Составы групп и задания (см. табл. 1) остаются из первой лабораторной работы.

Необходимо настроить использование системы контроля версий *git* всей группой разработчиков проекта (можно использовать *GitLab*, *Bitbucket* и *GitHub*). Создать дополнительную ветку *dev* на основе *master*, в которую необходимо добавлять все новые изменения (если необходимо, то можно добавлять еще ветки, но ответвлять их от ветки *dev*).

Создать файл *readme.md*, в котором будет содержаться описание разрабатываемого проекта (можно подкорректировать ТЗ из первой лабораторной работы). Затем закоммитить изменения в *readme* файле в *git* репозиторий.

Примечание. Коммитить только рабочий функционал. Вся разработка ведется в ветку *dev* и ответвленные от нее ветки. Мержить в ветку мастер только рабочие наработки. На каждом этапе разработки должна быть рабочая версия программы с некоторым набором функционала.

Туториал по использованию *git* смотреть на сайте https://githowto.com/ru.

Таблица 1 – Варианты заданий

Вариант	**Задание**
1	Змейка
2	Тетрис
3	Крестики-нолики
4	Пакман
5	Гоночки
6	Танчики
7	Pocket tanks
8	Worms
9	Морской бой
10	Bomberman
11	Теннис 2D
12	Воллейбол 2D
13	Шашки
14	Косынка
15	Свинка (Карточная игра. Выигрывает тот, кто откроет больше парных карт)
16	Астероиды
17	Fluffy Bird

Контрольные вопросы

1. Для чего нужны системы контроля версий.
2. Для чего нужны commit, push, pull.
3. Для чего нужны ветки (branch).
4. Для чего нужен pull request.

ЛАБОРАТОРНАЯ РАБОТА №3

«ЗНАКОМСТВО И СОЗДАНИЕ ПРОЕКТА»

Цель работы: изучить подходы проектирования и создания структуры проекта.

ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Шаблон проектирования или паттерн – повторяемая архитектурная конструкция, представляющая собой решение проблемы проектирования в рамках некоторого часто возникающего контекста.

Любой паттерн описывает задачу, которая снова и снова возникает в нашей работе, а также принцип ее решения, причем таким образом, что это решение можно потом использовать миллион раз, ничего не изобретая заново.

В общем случае паттерн состоит из четырех основных элементов:

Имя. Сославшись на него, мы можем сразу описать проблему проектирования; ее решения и их последствия. Присваивание паттернам имен позволяет проектировать на более высоком уровне абстракции.

Задача. Описание того, когда следует применять паттерн.

Решение. Описание элементов дизайна, отношений между ними, функций каждого элемента.

Результаты – это следствия применения паттерна и разного рода компромиссы.

Типы шаблонов проектирования [5-7]:

– порождающие паттерны;

– структурные паттерны;

– поведенческие паттерны.

Порождающие паттерны [5-7]:

1) Абстрактная фабрика – семейства порождаемых объектов.
2) Одиночка – единственный экземпляр класса.
3) Прототип – класс, из которого инстанцируется объект.
4) Строитель – способ создания составного объекта.
5) Фабричный метод – инстанцируемый подкласс объекта.

Структурные паттерны [6, 7]:

1) Адаптер – объект, обеспечивающий взаимодействие двух других объектов, один из которых использует, а другой предоставляет несовместимый с первым интерфейс.
2) Компоновщик – объект, который объединяет в себе объекты, подобные ему самому.
3) Декоратор – класс, расширяющий функциональность другого класса без использования наследования.

4) Фасад – объект, который абстрагирует работу с несколькими классами, объединяя их в единое целое.

Поведенческие паттерны [6, 7]:

1) Команда – представляет действие, объект команды заключает в себе само действие и его параметры
2) Интерпретатор – решает часто встречающуюся, но подверженную изменениям, задачу.
3) Наблюдатель – определяет зависимость типа «один ко многим» между объектами таким образом, что при изменении состояния одного объекта все зависящие от него оповещаются об этом событии.

Задание.

Составы групп и задания (см. табл. 1) остаются из предыдущей лабораторной работы.

Необходимо разработать структуру проекта (проект должен быть гибким и расширяемым) с использованием паттернов проектирования.

Таблица 1 – Варианты заданий

Вариант	**Задание**
1	Змейка
2	Тетрис
3	Крестики-нолики
4	Пакман
5	Гоночки
6	Танчики
7	Pocket tanks
8	Worms
9	Морской бой
10	Bomberman
11	Теннис 2D
12	Воллейбол 2D
13	Шашки
14	Косынка
15	Свинка (Карточная игра. Выигрывает тот, кто откроет больше парных карт)
16	Астероиды
17	Fluffy Bird

Контрольные вопросы

1. Что такое паттерн проектирования
2. Порождающие паттерны проектирования
3. Структурные паттерны проектирования
4. Поведенческие паттерны проектирования

ЛАБОРАТОРНАЯ РАБОТА №4

«РАБОТА С СИСТЕМОЙ ОТСЛЕЖИВАНИЯ ОШИБОК»

Цель работы: разработать модульные тесты и настроить логирование с использованием *Log4net* (при разработке на *Java* использовать *Log4j* или любой другой логер).

ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Разработка через тестирование (англ. *test-driven development*, TDD) – техника разработки программного обеспечения, которая основывается на повторении очень коротких циклов разработки: сначала пишется тест, покрывающий желаемое изменение, затем пишется код, который позволит пройти тест, и под конец проводится рефакторинг нового кода к соответствующим стандартам.

Цикл разработки через тестирование

Приведенная последовательность действий основана на книге *Кента Бека* «Разработка через тестирование: на примере».

Добавление теста

При разработке через тестирование, добавление каждой новой функциональности (англ. feature) в программу начинается с написания теста. Неизбежно этот тест не будет проходить, поскольку соответствующий код ещё не написан. (Если же написанный тест прошёл, это означает, что либо предложенная «новая» функциональность уже существует, либо тест имеет недостатки.) Чтобы написать тест, разработчик должен чётко понимать предъявляемые к новой возможности требования. Для этого рассматриваются возможные сценарии использования и пользовательские истории. Новые требования могут также повлечь изменение существующих тестов. Это отличает разработку через тестирование от техник, когда тесты пишутся после того, как код уже написан: она заставляет разработчика сфокусироваться на требованиях до написания кода – тонкое, но важное отличие.

Запуск всех тестов: убедиться, что новые тесты не проходят

На этом этапе проверяют, что только что написанные тесты не проходят. Этот этап также проверяет сами тесты: написанный тест может проходить всегда и соответственно быть бесполезным. Новые тесты должны не проходить по объяснимым причинам. Это увеличит уверенность (хотя не будет гарантировать полностью), что тест действительно тестирует то, для чего он был разработан.

Написать код

На этом этапе пишется новый код так, что тест будет проходить. Этот код не обязательно должен быть идеален. Допустимо, чтобы он проходил тест

каким-то неэлегантным способом. Это приемлемо, поскольку последующие этапы улучшат и отполируют его.

Важно писать код, предназначенный именно для прохождения теста. Не следует добавлять лишней и, соответственно, не тестируемой функциональности.

Запуск всех тестов: убедиться, что все тесты проходят

Если все тесты проходят, программист может быть уверен, что код удовлетворяет всем тестируемым требованиям. После этого можно приступить к заключительному этапу цикла.

Рефакторинг

Когда достигнута требуемая функциональность, на этом этапе код может быть почищен. Рефакторинг – процесс изменения внутренней структуры программы, не затрагивающий её внешнего поведения и имеющий целью облегчить понимание её работы, устранить дублирование кода, облегчить внесение изменений в ближайшем будущем.

Повторить цикл

Описанный цикл повторяется, реализуя всё новую и новую функциональность. Шаги следует делать небольшими, от 1 до 10 изменений между запусками тестов. Если новый код не удовлетворяет новым тестам или старые тесты перестают проходить, программист должен вернуться к отладке. При использовании сторонних библиотек не следует делать настолько небольшие изменения, которые буквально тестируют саму стороннюю библиотеку, а не код, её использующий, если только нет подозрений, что библиотека содержит ошибки.

Модульное тестирование (англ. *unit testing*) – процесс в программировании, позволяющий проверить на корректность единицы исходного кода, наборы из одного или более программных модулей вместе с соответствующими управляющими данными, процедурами использования и обработки.

Идея состоит в том, чтобы писать тесты для каждой нетривиальной функции или метода. Это позволяет достаточно быстро проверить, не привело ли очередное изменение кода к регрессии, то есть к появлению ошибок в уже оттестированных местах программы, а также облегчает обнаружение и устранение таких ошибок.

Преимущества модульного тестирования

Цель модульного тестирования – изолировать отдельные части программы и показать, что по отдельности эти части работоспособны.

Этот тип тестирования **обычно выполняется программистами**.

Поощрение изменений

Модульное тестирование позже позволяет программистам проводить рефакторинг, будучи уверенными, что модуль по-прежнему работает корректно (регрессионное тестирование). Это поощряет программистов к изменениям

кода, поскольку достаточно легко проверить, что код работает и после изменений.

Упрощение интеграции

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

Документирование кода

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

Отделение интерфейса от реализации

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

Логи (лог-файлы) – это файлы, содержащие системную информацию работы сервера или компьютера, в которые заносятся определенные действия пользователя или программы. Иногда также употребляется русскоязычный аналог понятия – журнал.

Их предназначение – протоколирование операций, выполняемых на машине, для дальнейшего анализа администратором. Регулярный просмотр журналов позволит определить ошибки в работе системы в целом, конкретного сервиса или сайта (особенно скрытые ошибки, которые не выводятся при просмотре в браузере), диагностировать злонамеренную активность, собрать статистику посещений сайта.

Поскольку основное программное обеспечение, установленное на сервере, чаще всего производит запись в системные журналы, то для каждой из таких программ будет свой журнал. В частности, можно выделить такие наиболее распространенные логи:

– основной файл лога (общая информация – данные о действиях с ядром системы, работе FTP-сервисов, DNS-сервера, файервола);

– лог загрузки системы (помогает выполнить отладку системы в случае, если она не загружается, сохраняет основные системные события (например, сбои оборудования));

– логи веб-сервера (данные об обращениях к серверу, информация об ошибках веб-сервера);

– логи сервера баз данных (запросы к базам данных, ошибки сервера);

– логи хостинговой панели, через которую осуществляется управление сайтом на хостинге (попытки входа в панель, обновления лицензии и панели, статистика использования ресурсов сервера);

– логи почтового сервера (записи о всех отправленных и доставленных сообщениях, ошибки почтового сервера, причины отклонения писем);

– логи планировщика задач — cron (протоколирование выполнения задач, ошибок при запуске крона).

Задание.

Составы групп и задания (см. табл. 1) остаются из первой лабораторной работы.

Необходимо разработать модульные тесты для проекта и настроить систему записи логов в файл с использованием технологии *Log4net* (при разработке на *Java* использовать *Log4j* или любой другой логер).

Таблица 1 – Варианты заданий

Вариант	**Задание**
1	Змейка
2	Тетрис
3	Крестики-нолики
4	Пакман
5	Гоночки
6	Танчики
7	Pocket tanks
8	Worms
9	Морской бой
10	Bomberman
11	Теннис 2D
12	Воллейбол 2D
13	Шашки
14	Косынка
15	Свинка (Карточная игра. Выигрывает тот, кто откроет больше парных карт)
16	Астероиды
17	Fluffy Bird

Контрольные вопросы

1. Что такое модульное тестирование.
2. Для чего нужно логирование.
3. Что такое *mock*-объект.
4. Что такое TDD.
5. Для чего нужны тесты.

ЛАБОРАТОРНАЯ РАБОТА №5

«СОЗДАНИЕ И КОНФИГУРАЦИЯ АВТОМАТИЧЕСКОЙ СБОРКИ»

Цель работы: изучить способы создания и конфигурации автоматической сборки.

ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Автоматизация сборки – этап написания скриптов или автоматизация широкого спектра задач, применяемого разработчиками в их повседневной деятельности. Включает в себя такие действия, как:

– компиляция исходного кода в бинарный код

– сборка бинарного кода;

– выполнение тестов;

– разворачивание программы на производственной платформе;

– написание сопроводительной документации или описание изменений новой версии.

Продвинутая автоматизация сборки предоставляет возможность удаленному пользователю управлять обработкой распределённых сборок и/или распределённой обработкой сборки. Термин «распределённые сборки» подразумевает, что вызовы компилятора и линковщика могут передаваться множеству компьютеров для ускорения скорости сборки. Данный термин часто путают с «распределённой обработкой». Распределённая обработка означает, что каждый этап процесса может быть адресован разным машинам для выполнения ими данного шага. Например, этап после сборки может потребовать выполнения множества тестовых скриптов на множестве машин. Распределённая обработка позволяет послать команду на исполнение различных тестовых скриптов на разных машинах. Распределённая обработка – не то же самое, что и распределённая сборка! Распределённая обработка не может взять скрипты от *make* или *maven*, разбить их и послать команды на компиляцию и линковку различным машинам. Распределённый процесс сборки должен обладать определенной логикой, чтобы правильно определить зависимости в исходном коде для того чтобы выполнить этапы компиляции и линковки на разных машинах. Решение автоматизации сборки должно быть способно управлять этими зависимостями, чтобы выполнять распределённые сборки. Некоторые инструменты сборки могут распознавать подобные взаимосвязи автоматически (*Rational ClearMake distributed*, *Electric Cloud ElectricAccelerator*), а другие зависят от пользовательских указаний (*Platform LSF lsmake*) Автоматизация сборки, способная рассортировывать взаимосвязи зависимостей исходного кода, также может быть настроена на выполнение действий компиляции и линковки в режиме параллельного выполнения. Это означает, что компиляторы и линковщики могут быть вызваны в многопоточном режиме на машине, сконфигурированной с учётом наличия более одного процессорного ядра.

MSBuild – платформа сборки проекта, разработанная *Microsoft*. Обычно применяется в сочетании с *Visual Studio*. *MSBuild* версии 2.0 является частью

.NET Framework 2.0 и предназначен для работы с *Visual Studio* 2005. Версия *MSBuild* 3.5, объединённая с *.NET* 3.5 (и *Visual Studio* 2008), позволяет собирать проекты с поддержкой *.NET* версий 2.0, 3.0 или 3.5 на выбор (т. н. «многоцелевая сборка», или *multi-targeting*).

Так как *MSBuild* доступен в составе *.NET*, то можно собирать проекты и решения *Visual Studio* без установленной *IDE Visual Studio*. *MSBuild* не требует дополнительных денежных отчислений.

MSBuild обрабатывает специальные файлы проекта *MSBuild*, имеющие схожий с *Apache Ant* и *NAnt XML*-синтаксис. Несмотря на то, что синтаксис базируется на правильно оформленной (*well-defined*) *XML*-схеме, основная структура и обработка схожа с традиционной *Unix*-утилитой *Make*: пользователь указывает исходные файлы (как правило, это файлы с исходным кодом) и что должно получиться в результате (обычно – готовое приложение), а утилита сама решает, что и в каком порядке нужно делать.

Не все инструменты автоматизации сборки могут выполнять распределённые сборки. Большинство из них лишь реализует поддержку распределённой обработки. Кроме того, большинство решений, поддерживающих распределённые сборки, могут лишь обрабатывать код на языках Си и C++. Решения автоматизации сборки, поддерживающие распределённую обработку, зачастую основаны на *Make* и не поддерживают *Maven* или *Ant*.

В качестве примера решения распределённой сборки можно привести *Xoreax's IncrediBuild* для платформы *Microsoft Visual Studio*. Это может потребовать специфической настройки программного окружения чтобы успешно функционировать на распределённой платформе (нужно указать расположение библиотек, переменные окружения и т. д.).

Более подробная информация по *MSBuild* написана здесь: https://msdn.microsoft.com/en-us/library/dd393574.aspx, https://msdn.microsoft.com/en-us/library/ms164311.aspx .

Пример конфигурации проекта в *cmd*.

- В Path должен быть прописан путь к *.Net Framework* если этого нет, то команда *MSBuild* не сработает.

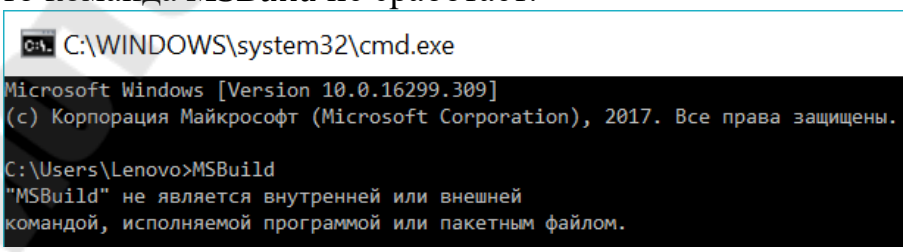

Рисунок 5.1 – Запуск MSBuild с не прописанным путем к *.NET Framework*

- Можно прописывать путь к фреймворку, но тогда MSBuild нужно запускать из папки с фреймворком (%SystemRoot%/Microsoft.NET/Framework/v4.0).

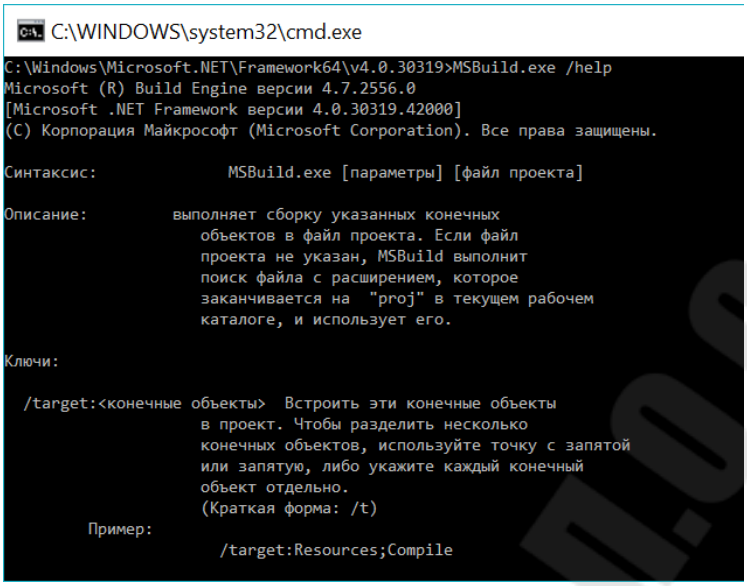

Рисунок 5.2 – Запуск *MSBuild* без прописывания пути к *Path*

Конфигурационными файлами проекта для *MSBuild* являются файлы с расширением **.cproj* (для проектов на *C#*). Данные файлы создаются автоматически IDE Пример такого файла представлен на рисунке 5.2.

Рисунок 5.3 – Конфигурационный файл для *MSBuild*

Задание.

Составы групп и задания (см. табл. 1) остаются из первой лабораторной работы.

В зависимости от выбранной платформы для разработки (*Java* или *.NET*) настроить конфигурацию сборки:

– Для *.NET*: настроить сборку через MSBuild. Написать bat-файл, в котором через параметры можно передать путь к проекту для сборки и тип конфигурации (DEBUG или RELEASE). В bat-файле сразу должен выполняться ребилд (rebuild) проекта под указанную конфигурацию.

– Для *Java*: настроить сборку проекта с использованием *Gradle* или *Maven* (настроить сборку проекта *jar*-файл).

Таблица 1 – Варианты заданий

Вариант	**Задание**
1	Змейка
2	Тетрис
3	Крестики-нолики
4	Пакман
5	Гоночки
6	Танчики
7	Pocket tanks
8	Worms
9	Морской бой
10	Bomberman
11	Теннис 2D
12	Воллейбол 2D
13	Шашки
14	Косынка
15	Свинка (Карточная игра. Выигрывает тот, кто откроет больше парных карт)
16	Астероиды
17	Fluffy Bird

Контрольные вопросы

1. Что такое автоматическая сборка.
2. Для чего нужны конфигурационные файлы.
3. Зачем нужно настраивать автоматическую сборку.

Список использованных источников

1. Jalote, P. Software Engineering. A Precise Approach = П. Инженерия программного обеспечения. Точный подход / P. Jalote. – Indiy: New Delhi, 2010. – 301 p.
2. Благодатских, В.А. Стандартизация разработки программных средств: учебное пособие для вузов / В.А. Благодатских, В.А. Волнин, К.Ф. Поскакалов; под ред. О.С. Разумова. – Москва: Финансы и статистика, 2006. – 285с.
3. Орлов, С. А. Программная инженерия: технологии разработки программного обеспечения / С. А. Орлов. - 5-е изд., обновл. и доп. – Санкт-Петербург [и др.]: Питер, 2017. – 640 с.
4. Пайлон, Д. Управление разработкой ПО / Дэн Пайлон, Расс Малз; [перевел с англи. В. Шрага]. – Санкт-Петербург [и др.]: Питер, 2014. – 459 с.
5. Приемы объектно-ориентированного проектирования. Паттерны проектирования / Э. Гамма [и др.]; пер. с англ. А. Слинкина. - Санкт-Петербург [и др.]: Питер, 2014. – 366 с.
6. Приемы объектно-ориентированного проектирования. Паттерны проектирования / Э. Гамма [и др.]. – Санкт-Петербург [и др.]: Питер, 2017. – 366 с.
7. Фаулер, М. Шаблоны корпоративных приложений / М. Фаулер. – М. «Вильямс», 2012. – 544 с.