Министерство науки и высшего образования Российской Федерации

ФГАОУ ВО «УрФУ имени первого Президента России Б.Н. Ельцина»

Институт радиоэлектроники и информационных технологий-РТФ

Пояснительная записка к проекту по теме:

**Автоматизированное модульное тестирование блока игры «Лукоморье»**

Преподаватель Присяжный Алексей Владимирович

**ФИО преподавателя**

Студент Пех Александра Романовна

**ФИО студента**

Специальность (направление подготовки)

09.04.04 Программная инженерия

Группа РИМ-120990

Оглавление

[Введение 3](#_Toc125125716)

[Цель проекта 4](#_Toc125125717)

[Что можно тестировать 4](#_Toc125125718)

[Как пишутся тесты 5](#_Toc125125719)

[Какими должны быть тесты 5](#_Toc125125720)

[Плюсы тестирования 6](#_Toc125125721)

[Разработка в рамках игры 7](#_Toc125125722)

[Travis CI 7](#_Toc125125723)

[Заключение 8](#_Toc125125724)

[Библиографический список 8](#_Toc125125725)

# Введение

Игра “Лукоморье” происходит в средневековой (альтернативной) Руси, также в этом мире существует магия и мистические существа. Главная фабула проекта - – сохранение мира.

Сюжет:

Важная роль – сохранение баланса между силами Света и Тьмы. За сохранение равновесия отвечает Тур. Его рога оплетают кольца Силы. Однако, одна из его подков сломалась, Тур больше не может сохранять равновесие. Это привело к нарушению баланса. Мир стал погружаться во тьму. Герою необходимо помочь восстановить баланс сил, иначе мир погибнет в Хаосе.

Ученый Кот рассказывает легенду о Кузнеце, способном восстановить подкову хранителя равновесия. Чтобы найти Кузню необходимо собрать 4 фрагмента карты. Они хранятся у Водяного, змея Горыныча, Лихо и Кощея. У Кота есть портал, ведущий в лес к Лихо.

В качестве подарка Кот отдает Веретено – в минуту наивысшей опасности оно вернёт Героя в прошлое, в момент встречи с Котом, поэтому при «гибели» игрок возвращается в Лукоморье к Коту.

При получении 4 фрагментов карты отрывается концовка – перемещение к Кузнецу, решение проблемы, мир спасен.

“Лукоморье” – казуальная roguelike игра в пиксельном стиле с видом сверху. Игроку необходимо пройти по 4 процедурно сгенерированным локациям и сразиться с различными противниками. В конце каждой локации есть уникальный Босс. Убивая противников, герой получает монеты и рандомную экипировку становясь сильнее.

Основная задача игрока – прохождение сюжета полагаясь на свои навыки, полученные благодаря предыдущим попыткам пройти игру, а также удачно выпавшие предметы. Игрок основывает свой способ прохождения в зависимости от выпавших предметов, пытаясь грамотно их использовать в различных ситуациях.

Игровой процесс заключается в управлении «персонажем» для прохождения карты и сражений.

# Цель проекта

Обычно после написания кода его проверяют. Если речь о какой-то функции, то можно написать простой скрипт, который будет вызывать ее с разными аргументами, и смотреть, что она вернет. Если вы сделали сайт или приложение, то вы открываете его, жмете ссылки и кнопки, проверяете что все отображается верно. Это называется ручное тестирование или QA (Quality Assurance — контроль качества) — человек проверяет работу программы. Если мы попробуем автоматизировать этот процесс, и написать программу, которая проверяет правильность другой программы, то это называется автоматизированное тестирование.

Главный плюс автоматических тестов — то, что они выполняются намного быстрее, чем ручное тестирование, и вам не надо тратить на это свое время (или время тестировщика). Это позволяет запускать их хоть после каждого изменения в коде.

Также, тесты позволяют «защитить» написанный код. Если кто-то в команде (или вы сами) нечаянно «сломал» ваш код, тесты это обнаружат и укажут, что именно перестало работать. Поэтому править код становится комфортнее и спокойнее — не надо бояться, что в ходе правки вы сломали какой-то функционал и не заметили. Тестирование особенно полезно при разработке сложных приложений в большой команде.

# Что можно тестировать

Тестировать программу можно на разных уровнях: код (юнит-тесты и интеграционное тестирование), API (если оно есть) и GUI (интерфейс пользователя). Разные виды тестов лучше подходят в разных ситуациях.

Модульное тестирование (Unit Testing) – это тип тестирования программного обеспечения, при котором тестируются отдельные модули или компоненты программного обеспечения. Его цель заключается в том, чтобы проверить, что каждая единица программного кода работает должным образом. Данный вид тестирование выполняется разработчиками на этапе кодирования приложения. Модульные тесты изолируют часть кода и проверяют его работоспособность. Единицей для измерения может служить отдельная функция, метод, процедура, модуль или объект.

Это значит, что если код обращается к каким-то другим классам, то вместо них ставятся классы-заглушки. Если код обращается к файлам, базе данных, по сети, то это все тоже заменяется на заглушки, возвращающие заранее подготовленные данные. Это делается потому, что в юнит-тестировании мы тестируем именно одну функцию, а не правильность работы базы данных, жесткого диска или удаленного сервера.

Обычно юнит-тест передает функции разные входные данные и проверяет, что она вернет ожидаемый результат. Юнит-тесты хорошо тестируют такой 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 нужно большое число заглушек.

Ради ускорения выполнения тестов, обычно используют базу данных, хранящую данные в памяти, а не на диске (MySQL умеет это).

Если проводить аналогии, например, с тестированием авиадвигателя, то юнит-тесты - это тестирование отдельных деталей, клапанов или заслонок.

Тестировать можно не любой код. Если в вашем коде жестко прописаны параметры соединения с базой данных или пути к папкам без возможности их поменять, вы вряд ли сможете использовать для тестов временную БД. То же самое, если классы в вашем коде сильно связаны, и вы не используете dependency injection, если используются глобальные переменные или везде статические методы.

# Как пишутся тесты

Составление тестов начинается со сбора списка требований к тестируемому коду. Затем для каждого требования пишется тестовый сценарий, который проверяет, что оно выполняется. Сценарии бывают позитивными — когда мы передаем корректные данные и ждем успешный результат, и негативными — когда мы передаем неправильные данные и ждем сообщение об ошибке.

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.

Поэтому на каждое требование мы пишем отдельный тест - это позволит при ошибке понять, что именно сломалось. Тесты обычно пишут в стиле Arrange, Act, Assert.

* часть Arrange — здесь производится создание и инициализация требуемых для проведения теста объектов
* часть Act — собственно проведение тестируемого действия
* часть Assert — здесь производится сравнение полученного результата с эталонным

Тесты обычно пишут не с нуля, а с использованием фреймфорка.

# Какими должны быть тесты

Есть список требований к созданию тестов.

1. Маленький объем - идеальный тест укладывается в 5-15 строк и проверяет только одно требование. Он должен быть легко читаемым, чтобы с первого взгляда был очевиден сценарий, по которому происходит тестирование.
2. Повторяемость - тест должен выдавать одинаковый результат при каждом запуске. Стоит избегать использования случайных величин, так как иначе может получиться «нестабильный» тест, который то работает, то нет, и каждый раз надо тратить время, и искать причину.
3. Тест не должен использовать тот же алгоритм, что и проверяемый код, так как можно сделать одинаковую ошибку и в коде, и в тесте, и не заметить её. Например, если мы тестируем функцию решения уравнения, то проверять ее можно подстановкой решения обратно в уравнение.

Тесты должны выполняться в контролируемом окружении. Надо тестировать и позитивные, и негативные сценарии. Прежде чем писать тесты, надо составить план тестирования (что мы тестируем и как). Мы сначала определяем, какие возможности предоставляет приложение, а потом для каждой пишем несколько тест кейсов.

Тесты должно быть легко запустить, в идеале одной командой. Если для его запуска надо выполнить много действий, то людям будет лень это делать. В компаниях обычно настраивают CI сервер, который сам выкачивает обновления из репозитория, запускает тесты, и рассылает разработчикам сообщения при ошибках.

Обычно для тестов создают папку с названием tests в корне проекта.

Настройки для phpunit также можно задать не опциями, а в XML-файле phpunit.xml. Формат файла описан в мануале. Это удобнее, так как их в этом случае не надо печатать в командной строке

Вот полный примерный план 1 пункт = 1 тест.

Загрузка файла

* Загрузить файл и убедиться, что он загрузился
* Отправить пустую форму без файла
* Загрузить слишком большой файл

Страница файла

* Загрузить файл и убедиться, что он доступен для скачивания. Скачиваемый файл совпадает с загруженным
* Убедиться, что выводится правильная информация о файле

Медиаданные

* Загрузить картинку и убедиться, что превью выводится и она совпадает с картинкой
* Загрузить картинку неподдерживаемого типа вроде tif/bmp
* Загрузить аудиофайл и убедиться, что доступен плеер
* Загрузить видеофайл и увидеть плеер

Комментарии

* Отправить комментарий
* Отправить пустой комментарий
* Отправить комментарий-ответ

В качестве базы для тестов стоит использовать in-memory mysql базу.

# Плюсы тестирования

* Основная ветка всегда в рабочем состоянии (или мы быстро узнаем, что «что-то» не так)
* Нет ошибок в редакторе
* История сборки (можно найти, когда появился баг)
* Взаимодействие модулей
* Бизнес-логика
* Меньше неожиданностей
* Работает на любой стадии проекта

# Разработка в рамках игры

Одной из локаций в игре является «Магазин Яги». Магазин – особая локация которая встречается 1 раз на каждой локации. В магазине можно приобрести различные предметы за золотые монеты или решив головоломку. Мини-игра представляет собой тест с выбором правильного ответа. Для успешного прохождения мини-игры все тестовые задания должны быть решены верно.

Так возникает необходимость создания файла с вопросами, вариантами ответа и верным ответом. Так как проект разрабатывается командой сразу в нескольких направлениях сразу определить структуры мини-игры невозможно. Поэтому было принято решение, что все тестовые вопросы и ответы будут храниться в отдельном файле. Так при необходимости модернизации игры можно менять\добавлять\изменять существующий банк вопросов, а на код обработки — это не должно никак влиять.

Так удалось сформулировать важное и ключевое свойство кода обработки мини-игры в «Магазине» - не должно быть прямой связи с вопросом.

Ход работы:

Для написания прототипа приложения «Мини-игра Магазина Яги» для тестирования на языке программирования С++ были созданы два текстовых файла. В файл под именем test.txt запишем вопросы к тесту. Каждый вопрос в файле находится на двух строчках и состоит из вопроса и вариантов ответа. Во втором файле, назовём его answer.txt, в каждой строчке пропишем ответ на вопрос файла test.txt, выполняя заранее установленную последовательность.

При сохранении файлов важно установить кодировку символов ANSI для корректного отображения текста в консоли.

Сама обработка теста написана на С++.

Функция void test() читает из файла test.txt вопросы к тесту и выводит их в консоль. Открывает новый файл myanswer.txt и записывает в него ответы пользователя. По окончанию тестирования закрывает оба файла.

Функция result() открывает файлы answer.txt и myanswer.txt, сравнивает ответы пользователя с правильными ответами в файле answer.txt и выводит результат на экран.

# Travis CI

Чтобы не запускать тесты каждый раз руками, используют CI Server вроде Teamcity, Jenkins или Hudson. Но эти системы надо устанавливать и настраивать, в то время как Travis это сервис, который сам будет подсоединяться к твоему гитхаб репозиторию, брать оттуда изменения и прогонять тесты. Для open source проектов на github Travis CI бесплатен.

Travis CI — это continuous integration сервис для проектов на Github. Когда вы коммитите что-то в репозиторий, Travis CI может автоматически выполнять разные полезные действия. Например, он может запускать модульные тесты.

Ход работы:

1. Активируйте свои репозитории GitHub
2. Выберите план (бесплатная пробная версия или платный)
3. Добавьте файл .travis.yml в свой репозиторий.
4. Запустите свою первую сборку

При использовании Travis CI для работы с «Мини-игрой магазина Яги» результаты имеют дальнейший потенциал. Так как CI-сервер не только обеспечивает репозиторий артефактов для хранения результатов каждой сборки, чтобы при необходимости их можно было развернуть, но и может хранить и регулировать параметры для каждой среды в пайплайне. В дальнейшем можно указать, должны ли ваши скрипты развертывания автоматически срабатывать в зависимости от результатов, полученных на предыдущем этапе.

# Заключение

На текущий момент проект находится в стадии разработки. Так как в проекте задействовано много людей – игра периодически падает. Поэтому введение модульного тестирования является важным моментом в решении проблем компиляции разных блоков.

К сожалению, пока нет возможности запустить проверку вместе всех блоков автоматически и приходится тестировать каждый момент полуавтоматически.

При работе с одним авто тестом программа работает корректно и быстро, что важно при динамической разработке игры.

Использование CI-сервера позволит добавить автоматическую работу дополнительных скриптов в зависимости от результатов, полученных на предыдущем этапе. Благодаря этому в игру можно добавить дополнительный блоки для более сложной и интересной структуры.

# Библиографический список

1. Проект «Лукоморье». Дизайн-документ
2. Статья «Автоматизированное тестирование» - <https://gist.github.com/yention/285ac408e08a0f419d71463f3b1d86b8>
3. Статья «Модульное тестирование» - <https://coderlessons.com/tutorials/kachestvo-programmnogo-obespecheniia/ruchnoe-testirovanie/modulnoe-testirovanie-2>
4. Официальный сайт - программное обеспечение - <https://www.travis-ci.com/>