МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение
высшего образования
«ТЮМЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

ИНСТИТУТ МАТЕМАТИКИ И КОМПЬЮТЕРНЫХ НАУК
Кафедра программного обеспечения

РЕКОМЕНДОВАНО К ЗАЩИТЕ
В ГЭК И ПРОВЕРЕНО НА ОБЪЕМ
ЗАИМСТВОВАНИЯ
Заведующий кафедрой
к.т.н., доцент
М. С. Воробьева
24.06 2019 г.

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА
(магистерская диссертация)

ПРОЕКТИРОВАНИЕ И РАЗРАБОТКА ПРОГРАММНОГО МОДУЛЯ ДЛЯ ПОВЫШЕНИЯ КАЧЕСТВА ПРОЦЕССА ТЕСТИРОВАНИЯ ПЛАГИНА ДЛЯ ПЛАТФОРМЫ PETREL

02.04.03. Математическое обеспечение и администрирование информационных систем
Магистерская программа «Разработка, администрирование и защита вычислительных систем»

Выполнила работу
Студентка 2 курса
очной формы обучения

Дериглазова
Анастасия
Олеговна

Руководитель работы
к. физ-мат. н., доцент
кафедры программного обеспечения

Ступников
Андрей
Анатольевич

Рецензент
Руководитель группы по
обеспечению качества программных
продуктов компании Schlumberger
Logelco Inc.

Чашкова
Анна
Дмитриевна

г. Тюмень, 2019

Оглавление

Введение 4

ГЛАВА 1. ОРГАНИЗАЦИЯ ТЕСТИРОВАНИЯ ПРОГРАММНЫХ ПРОДУКТОВ 6

1.1. Основные определения 6

1.2. Жизненный цикл тестирования 8

1.3. Классификация методов тестирования по степени автоматизации 9

1.3.1. Ручное тестирование 9

1.3.2. Автоматизированное тестирование 10

1.4. Классификация по знанию внутреннего устройства объекта 12

1.4.1. Тестирование черного ящика 12

1.4.2. Метод белого ящика 13

1.4.3. Метод серого ящика 13

1.5. Классификация методов тестирования по изолированности тестируемых объектов 14

1.5.1. Модульное тестирование 14

1.5.2. Интеграционное тестирование 17

1.5.3. Системное тестирование 18

1.6. Вывод по главе 1 20

ГЛАВА 2. Процесс тестирования программного продукта 21

2.1. Описание объекта тестирования 21

2.2. .NET Framework 25

2.3. Фреймворк NUnit 26

2.4. Среда тестирования Ocean API 27

2.5. Visual Studio Team System 27

2.6. Разработка набора тестов и подбор тестовых данных 29

2.7. Автоматизация тестов 37

2.8. Полученная выгода после внедрения автоматизации 44

Заключение 47

Список литературы 48

Приложение 1. Классы для тестирования создания нефтяного резервуара 50

Приложение 2. Класс для тестирования создания газового резервуара 52

Приложение 3. Класс для тестирования графика Energy plot 54

Приложение 4. Класс для тестирования графика Havlena Odeh 56

Приложение 5. Класс для инициализации тестовых параметров 58

Введение

В настоящее время многие компании занимаются разработкой программного обеспечения, и одной из актуальных профессий является программист. Но в процессе разработки необходимо также быть уверенными, что отсутствуют сбои при использовании продукта и он работает в соответствии с ожиданиями заказчика, этим занимается инженер по тестированию.

Компания Schlumberger занимается не только нефтесервисными услугами непосредственно на месторождении, но и разрабатывает программное обеспечение для геологического и гидродинамического моделирования, проектирования скважин, сопровождения бурения, петрофизической интерпретации других направлений в области нефтегазовой добычи. В компании ведется разработка нескольких платформ, предназначенных для работы с геолого-геофизическими и петрофизическими моделями. Petrel и Techlog – одни из примеров таких платформ, для которых ведется разработка программных модулей, предоставляющих дополнительные расширенные функции и возможности при работе с этими платформами.

Каждый разрабатываемый продукт должен стабильно работать и соответствовать требованиям, которые были выдвинуты заказчиком. Для этого необходимо проводить различные виды тестирования: функциональное, приемочное, тестирование производительности и безопасности, а также многие другие. Обеспечением качества продуктов в процессе разработки занимаются специально обученные люди – инженеры по обеспечению качества программного обеспечения. Довольно часто инженерам по тестированию приходится выполнять одни и те же тесты большое количество раз, например, в процессе регрессионного тестирования

продукта. В связи с этим не только сильно возрастает вероятность ошибок из-за «человеческого фактора», но и тратится большое количество времени и ресурсов на выполнение тестовых сценариев для проверки функциональности продуктов.

В данный момент в Schlumberger разрабатывается плагин, предназначенный для гидродинамического моделирования, и поскольку перед его выпуском мне, как инженеру-тестировщику, необходимо провести качественное тестирование, то он был выбран в качестве объекта для написания автоматизированных тестов.

Таким образом, целью данной магистерской диссертации является автоматизация функционального тестирования плагина «Material Balance».

Задачи магистерской диссертации, на основе поставленной цели, следующие.

1. Изучить общие подходы и инструменты тестирования с целью применения их в выпускной квалификационной работе.
2. Изучить тестируемый продукт.
3. Разработать набор тестов и подготовить тестовые данные для функционального тестирования.
4. Автоматизировать тесты и проверить их работоспособность.
5. Оценить полученную выгоду после внедрения автоматизации.

ГЛАВА 1. ОРГАНИЗАЦИЯ ТЕСТИРОВАНИЯ ПРОГРАММНЫХ ПРОДУКТОВ

1.1. Основные определения

Тестирование программного обеспечения является важным этапом в жизненном цикле проекта. Рассмотрим основные термины, связанные с обеспечением качества программных продуктов.

Тестирование программного обеспечения представляет собой проверку, что программа предоставляет ожидаемый результат, осуществляемый на конечном наборе тестов, выбранных подходящим образом для этого домена.

Спецификация продукта – это соглашение между разработчиками команды по разработке программного обеспечения, которое определяет, какие функции должны быть у продукта [2].

В таблице 1 представлены основные термины, используемые при тестировании программного обеспечения.

Таблица 1. Основные термины, используемые в тестировании

Термин	Определение
Error	Ошибка пользователя, когда продукт используется не по назначению.
Bug или дефект	Ошибка члена команды по разработке программного продукта, когда он работает не так, как описано в Спецификации.
Failure	Сбой (программный или аппаратный) в работе программы или ее компонента.
Баг репорт (Bug report)	Документ, в котором описываются действия,

	которые привели программный продукт к некорректной работе, также в этом отчете указывается ожидаемый результат, тестируемая версия продукта и, возможно, причины.
Тестовый случай (Test case)	Артефакт, описывающий условия, параметры и конкретные шаги, для проверки работоспособности реализованного функционала или его части.
План тестирования (Test Plan)	Документ, содержащий в себе объем работ по тестированию продукта, необходимые критерии для начала и окончания тестирования, стратегии и необходимого оборудования.

Качество программного обеспечения определяет совокупность следующих характеристик:

1. Функциональность – способность программы исправно решать требуемые пользователем задачи.
2. Надежность – способность продукта соответствовать заданному уровню пригодности в течение определенного интервала времени.
3. Удобство использования – легкость для понимания и освоения программы пользователем.
4. Эффективность – способность программы обеспечить должный уровень производительности.
5. Удобство сопровождения – возможность тестирования, поддержки и реализации новых функций в программном обеспечении.
6. Портативность – возможность легко перенести программное обеспечение с одного окружения на другое.

1.2. Жизненный цикл тестирования

Жизненный цикл тестирования представляет собой замкнутую последовательность действий. Существует множество вариаций для описания жизненного цикла тестирования, однако, общая концепция остается неизменна, рассмотрим вариант, предлагаемый автором книги «Тестирование программного обеспечение. Базовый курс» [3].

1. Стадия 1. Вначале происходит анализ требований и планирование. На этой стадии можно получить такую информацию как: объем работы, что предстоит тестировать и необходимое время.
2. Стадия 2. На этой стадии формулируются критерии, когда можно начинать и заканчивать тестирование, а также возможности его приостановки и возобновления
3. Стадия 3 позволяет уточнить детали стратегии тестирования.
4. Стадия 4 посвящается разработке тест кейсов, дизайну тестовых сценариев и работе над прочими артефактами, которые будут использоваться в процессе тестирования продукта.
5. Стадия 5 и стадия 6. Эти две стадии неразрывно связаны между собой, поскольку на первой происходит выполнение тестовых сценариев, а на второй регистрация найденных ошибок. Так как фиксировать дефекты следует сразу же после обнаружения, то есть в процессе выполнения тест-кейсов.
6. Стадия 7 и стадия 8. Это анализ выполнения тестирования и анализ отчетности.

Стоит отметить, что жизненный цикл тестирования является частью жизненного цикла всего программного продукта, поэтому они неразрывно связаны друг с другом. Тестирование каждой из компонент продукта должно проводиться своевременно, так как цена и трудозатраты на исправление ошибок на более поздних стадиях разработки будут гораздо выше, чем могли бы быть в начале.

1.3. Классификация методов тестирования по степени автоматизации

1.3.1. Ручное тестирование

Ручное тестирование выполняется тестировщиком без использования каких-либо дополнительных программных средств. Часто этот вид выполняется с позиции пользователя, и допускается отходить от заранее подготовленного плана тестирования, так как это позволит провести максимально полное тестирование программного продукта.

Будет неверным утверждение, что ручное тестирование – это устаревший метод. Именно ручное тестирование является более гибким. Рассмотрим случаи, когда автоматизировать тесты нет необходимости [6].

1. Автоматизация не оправдывает затрат.
2. Сценарии не будут включаться в регрессионное тестирование или непрерывную интеграцию.
3. Тесты являются временными.
4. Автоматизация сценариев неоправданно сложная.
5. Команда разработки не работает по технологии «Test Driven Development».

Ручной подход хорошо подходит для исследовательского тестирования, где отсутствуют явные тест-кейсы, а специалисты по тестированию полагаются на свой опыт и знания.

1.3.2. Автоматизированное тестирование

Автоматизация тестирования представляет собой процесс проверки программного обеспечения, включающий запуск, инициализацию, анализ и выполнение программы и генерацию результатов, автоматически с помощью специальных инструментов.

Применение средств автоматизации будет актуально для специалистов по тестированию во многих случаях.

1. Длинные пользовательские сценарии.
2. Рутинные операции (например, заполнение большого количества полей в пользовательском интерфейсе).
3. Backend тестирование, в том числе работа с базами данных.
4. Работа с математическими формулами, где нужны точные расчеты.

Святослав Куликов в своей книге [3] говорит о случаях, в которых наиболее эффективна автоматизация. В таблице 2 приведены задачи и проблемы, которые автоматизация может решить.

Таблица 2. Виды тестирования

Вид тестирования / случай	Решаемая проблема
Модульное тестирование	Тестирование атомарных участков кода. Когда продукт довольно большой, классов и методов

	огромное количество и выполнить такое тестирование человеку вручную невозможно.
Интеграционное тестирование	Тестирование взаимодействия различных компонент программы.
Регрессионное тестирование	Проверка, с помощью которой необходимо удостоверится, что раннее разработанная функциональность работает и не была сломана за счет внедрения новой.
Тестирование с использованием методов комбинаторики	Многократное выполнение одних и тех же тест кейсов с различными значениями, а также генерация этих значений в соответствии с законами комбинаторики.
Дымовое тестирование	Проверка запуска и работоспособности системы. Как правило, тестовые случаи для такого тестирования достаточно легкие и их легко автоматизировать.
Тестирование производительности	Создание большой нагрузки на тестируемый продукт с большими объемами данных.
Программы без графического интерфейса	Приложения, не предназначенные для взаимодействия с человеком, и которые невозможно полностью проверить вручную.

За счет применения автоматизации значительно повышается качество тестирования, это связано с тем, что в первую очередь, исключается так называемый «человеческий фактор». Когда человек привыкает выполнять одни и те же действия, то может не заметить ошибок. Также автоматизированное тестирование значительно экономит время, поскольку достаточно запустить скрипт, который все отработает и сгенерирует отчет. Однако стоит понимать, что автоматизация не может решить всех проблем

поскольку, любой отчет все равно стоит проанализировать человеку, а компьютер лишь выполнить монотонную работу и покажет результат прохождения тестов.

В ходе выполнения задач, поставленных в данной работе, планируется разработка автоматизированных тест кейсов для проверки функционала плагина для платформы Petrel. В плагине требуется большой объем входных параметров, а также в нем присутствуют сложные математические расчеты, на проверку которых может быть затрачено неоправданно большое количество времени. Это одни из тех ситуаций, в которых рекомендуется применять автоматизированное тестирование, и когда оно будет оправдывать свои затраты.

1.4. Классификация по знанию внутреннего устройства объекта

1.4.1. Тестирование черного ящика

При тестировании черного ящика предполагается, что инженер по тестированию не знает и не имеет доступа к внутренним компонентам тестируемой системы, и, следовательно, знание ее реализации не требуется. Метод тестирования черного ящика основывается на анализе спецификации продукта, и все тестовые случаи не имеют отсылок к коду программы.

Тестирование черного ящика играет важную роль в тестировании ПО, оно помогает в общей проверке функциональности системы. Данный вид тестирования выполняется с начала жизненного цикла проекта, и специалистам по тестированию необходимо участвовать на этапе сбора и анализа требований клиентов [15].

Тестирование методом черного ящика проходит с точки зрения потребителя, и это является большим достоинством такого подхода, так как

конечным пользователем является он и необходимо, чтобы ему было удобно и понятно пользоваться продуктом. Следующий плюс метода заключается в том, что некоторые дефекты программы невозможно обнаружить с помощью тестирования «белого ящика» (описание метода будет представлено ниже). Так, например, если какая-то функциональность вообще не была реализована программистом, то это легче обнаружить с позиции пользователя, потому что с точки зрения программного кода все работает исправно. Однако, если спецификация продукта неполная и нечеткая, то возникают сложности с проектированием тест-кейсов.

1.4.2. Метод белого ящика

Основу метода тестирования белого ящика для специалиста по тестированию составляет знание о внутренней части система и доступ к программному коду. Иными словами тестируется не какой-либо рабочий процесс пользователя или его поведение, а конкретная часть кода, написанная разработчиком. Использование такого метода при тестировании позволяет найти скрытые ошибки и оптимизировать код.

Тестирование методом белого ящика сосредоточено на внутренней логике и структуре кода и может выполняться на всех уровнях разработки системы (модульном, интеграционном, системном)

Основной сложностью в тестировании методом белого ящика является то, что необходимо уметь разбираться в так называемом «чужом» коде.

1.4.3. Метод серого ящика

Метод серого ящика сочетает в себе элементы методов черного и белого ящиков. Специалист по тестированию, используя данный метод,

проводит проверку на основе данных, которые он знает о системе, то есть каким образом реализованы те или иные функции. Но при этом специалист смотрит на тестируемый объект с позиции пользователя. Таким образом, используя данный метод, получается спроектировать более сложные тестовые сценарии.

1.5. Классификация методов тестирования по изолированности тестируемых объектов

1.5.1. Модульное тестирование

Тестирование, которое выполняется на самом низком уровне, называется модульным или юнит тестированием. Главной целью модульного тестирования является получение работоспособного кода, затратив при этом минимум средств. Идеей такого тестирования является то, чтобы разработчик писал тесты даже на самые простые функции. Грамотное деление объекта на модули обеспечивает архитектура проекта, спроектированная в соответствии с объектно-ориентированным программированием. Функция каждого класса в программном коде становится более ясной и прозрачной при использовании модульного тестирования. Таким образом, очень быстро можно отследить, не привели ли очередные изменения в коде к регрессии.

С помощью модульного тестирования можно проверить не только функциональность и надежность продукта. На этом этапе очень важно протестировать такие качества системы как эффективность и сопровождаемость, поскольку их почти невозможно проверить на более высоких уровнях (или это будет стоить гораздо дороже).

Рисунок 1 иллюстрирует схему модульного тестирования, а именно, красным цветом выделены компоненты, которые подлежат тестированию

при использовании данного подхода. В модульном тестировании не проверяется функционал, относящийся к взаимодействию компонент (синие стрелки на рисунке 1) или к работоспособности программного продукта в целом.

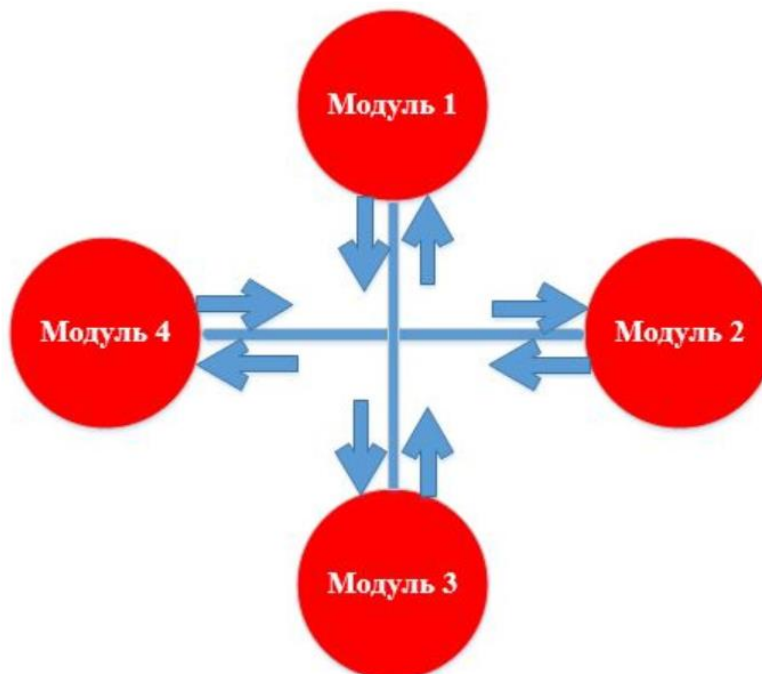

Рис. 1. Схема модульного тестирования

Юнит тестирование является первым уровнем из всей иерархии тестов, и, как правило, более многочисленным (Рис.2), а меньше всего тестов отводится на приемочное и ручное тестирование.

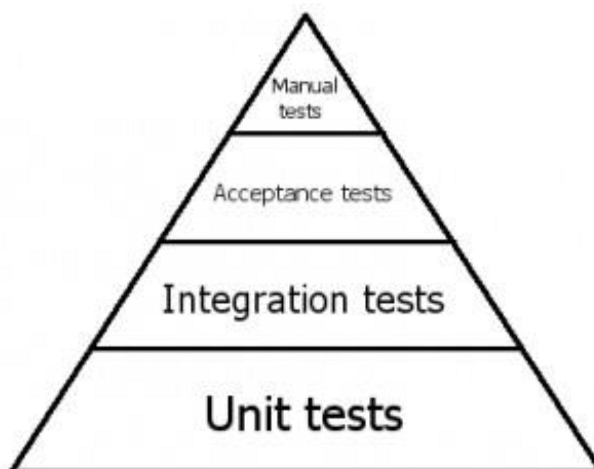

Рис. 2. Иерархия тестов

Автор книги «The art of Unit Testing with Examples in .NET» Roy Osherove [8] пишет, что самый распространённый и удобный метод проектирования модульных тестов – это паттерн AAA (Arrange – Act – Assert).

В секции Arrange следует конфигурировать в простом виде только необходимые данные. Здесь же нужно задать все тестовые данные и определить зависимости, причем у тестируемого класса может быть лишь один объект, и для всех зависимостей созданы двойники. Стоит отметить, что если произвести подмену невозможно, то это значит, что у класса сильная связность и необходимо провести рефакторинг.

Для упрощения кода первой секции модульных тестов, не стоит конфигурировать больше деталей, чем требуется для того, чтобы тест был сдан. Также большие объемы данных для прохождения тестов не требуются, и лучше ограничится минимальными. Автор книги «XUnit Test Patterns» Gerard Meszaros [9] считает, что вся конфигурация секции Arrange должна помещаться в тест, при этом, не обращаясь в сторонние классы и методы.

При работе в секции Act необходимо быть уверенными в том, что сторонние, нетестируемые методы не окажут влияния на состояние объекта и результаты, поэтому необходимо проводить тестирование вызова только одного метода. Тем не менее, необходимо вызывать еще конструктор, что в определенных случаях может спровоцировать проблему, когда неясно, по каким причинам тест не проходит. Конструктор выполняет компоновку объекта, а для правильного конструирования следует внедрить правильные зависимости. Марк Симан в своей книге «Внедрение зависимостей в .NET» [10] говорит, что зависимости лучше всего внедрять через конструктор, так как будет получен готовый, согласованный объект сразу же после вызова конструктора

1.5.2. Интеграционное тестирование

Интеграционное тестирование проводится для выявления дефектов в интерфейсах и взаимодействии между частями (модулями) программы, каждая из которых уж была проверена на этапе юнит тестирования [3]. Если при объединении нескольких модулей обнаруживается ошибка, это, скорее всего, связано с тем, как модули взаимодействуют. Интеграционное тестирование проверяет интеграцию модулей, интеграцию подсистем в одну единую систему и взаимодействие этих подсистем между собой. На Рис. 3. объект тестирования выделен красным цветом при использовании данного подхода.

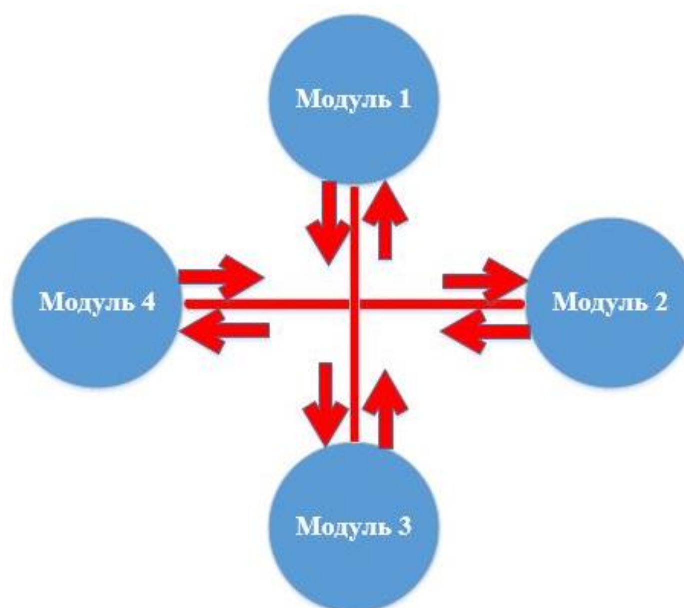

Рис. 3. Схема интеграционного тестирования

Довольно часто при объединении двух работающих модулей системы возникают проблемы. Поэтому наиболее труднорешаемые задачи связанны именно при совместной работе нескольких частей программы. Возникает вопрос – в каком порядке тестировать вместе компоненты? По факту объединение можно начинать тогда, когда протестированы хотя бы два любых модуля – это самое очевидное быстрое решение. Однако чем раньше

начинается интеграция, тем больше заглушек необходимо будет поставить, для того, чтобы протестировать эти части системы. В связи с этим менеджер на каждом проекте должен выбрать такую стратегию интеграции, которая будет оптимальной в конкретном случае по трудозатратам, времени и средствам.

Рассмотрим существующие подходы в интеграционном тестировании.

1. Снизу вверх. Модули собираются и тестируются с нижних уровней к вершине иерархии. Для тестирования таким подходом перед сборкой все модули должны быть готовы и протестированы по отдельности.
2. Сверху вниз. В этом подходе сборка происходит с более высоких модулей, для тех модулей, которые не включены в тест и находятся ниже по уровню, ставятся заглушки.
3. Большой взрыв. Тестируется сборка из всех модулей со всех уровней, что занимает меньшее количество времени, но требует больше внимательности к деталям.

1.5.3. Системное тестирование

После завершения интеграционного тестирования следующим уровнем является системное тестирование. Это полное тестирование программного обеспечения, выполняющееся для проверки соответствия заданным требованиям. Это тестирование необходимо проводить, несмотря на то, что перед этим были выполнены модульные и интеграционные тесты. Авторы книги «Software Testing Foundation» – Andreas Spillner, Tilo Linz, Hans Schaefer отмечают следующее [14]:

- на более низких уровнях тестирование проводится на основе технических характеристик, то есть с технической точки зрения

производителя ПО, системное же тестирование рассматривает продукт с точки зрения будущего пользователя;

- многие функции и характеристики системы являются результатом взаимодействия нескольких компонентов системы, соответственно, только при наличии всех этих компонентов можно удостоверится в корректности работы ПО.

К системному тестированию существует два подхода:

- на базе требований – когда тестовые случаи создаются на основе требований заказчика;
- на основе случаев использования – когда тестовые случаи проектируются на основе представлений пользовательского рабочего процесса.

Рисунок 4 иллюстрирует схему системного тестирования, на рисунке все объекты выделены красным, поскольку на таком уровне идет проверка программы в целом, а не отдельных ее компонент.

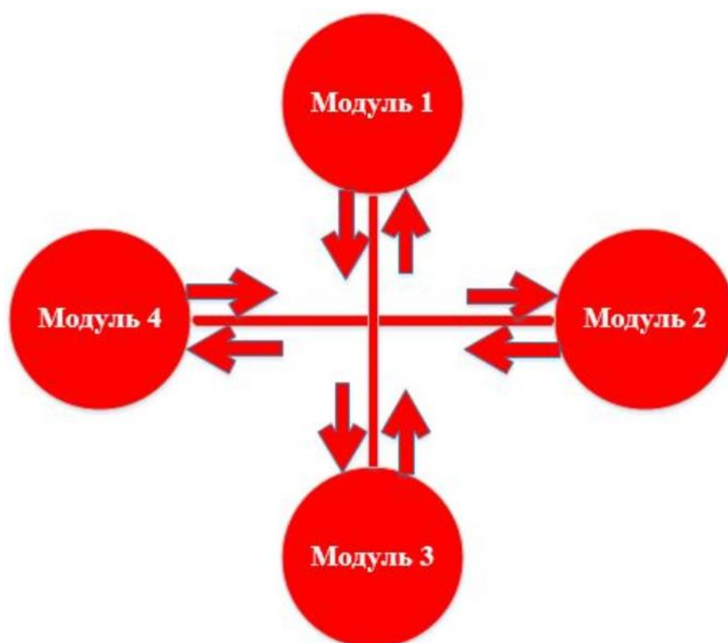

Рис. 4. Схема системного тестирования

1.6. Вывод по главе 1

В первой главе были рассмотрены несколько подходов и видов тестирования. В частности, целью магистерской диссертации является автоматизация тестов, поэтому были изучены два подхода к тестированию: ручной и автоматизированный, чтобы понять, в чем преимущества конкретного подхода и в каких случаях они применяются. При изучении соответствующей литературы было установлено, что автоматические тесты стоит создавать в случае, когда требуются точные расчеты или существует много рутинных действий, в процессе выполнения которых, человек может легко допустить ошибку. Дополнительно, автоматизированные тесты могут проигрывать ручному тестированию, когда их приходится часто изменять, поскольку это занимает гораздо больше времени, чем проведение тестирования вручную.

Также были рассмотрены уровни тестирования: модульное, интеграционное и системное. Понимание такого разделения важно, чтобы в тестах была четкая структура, и они были компактными и логичными, так как если один тест содержит в себе множество проверок, относящихся к разным уровням системы, то будет довольно сложно идентифицировать ошибку, в случае ее появления.

ГЛАВА 2. Процесс тестирования программного продукта

2.1. Описание объекта тестирования

Компания Schlumberger занимается разработкой программного обеспечения для различных направлений в области нефтегазовой добычи. Среди разрабатываемых продуктов присутствуют целые платформы, такие как: Petrel, Techlog, Studio, Ocean, Omega, Avoset. Для этих платформ в свою очередь разрабатываются программные модули, предоставляющие дополнительные расширенные функции и возможности при работе с этими платформами. Также ведется разработка прикладного программного обеспечения, например, семейство симуляторов ECLIPSE, INTERSECT, PIPESIM и другие.

Платформа Petrel предполагает расширение функционала за счет установки дополнительных модулей – плагинов. Их существует довольно много, все зависит от целей, которые преследует пользователь, как правило, один плагин выполняет какую-либо узкую функцию, например, моделирует гидроразрывы пласта, рассчитывает эффективность повышения нефтеотдачи, использует в рабочих процессах анализа подземных неопределенностей для скрининга, ранжирования и выбора геологических моделей и много другое.

В настоящее время для платформы Petrel ведется разработка плагина «Material Balance», который предназначен для поддержки технологических процессов разработки резервуаров. Для разработки автоматизированных тестов в данной работе был выбран плагин платформы Petrel.

Плагин «Material Balance» предоставляет инструмент для анализа и прогнозирования производительности нефтегазовых пластов, сочетая в себе

уравнение материального баланса и выбор модели водоносного горизонта для резервуара.

Плагин может использоваться для:

- обеспечения понимания производственных характеристик пласта;
- сопоставление истории данных пластового давления;
- определения механизмов, вызывающих истечение нефти и газа из пласта;
- определения начального объема флюида;
- проверки статических моделей.

В плагине присутствует возможность создания нефтяных и газовых резервуаров. Все нефтяные резервуары имеют следующие доменные объекты:

- tank (объект с описанием резервуара);
- график Havlena Odeh;
- график Energy;

Доменные объекты газового резервуара представлены ниже:

- tank (объект с описанием резервуара);
- график P/Z;
- график Havlena Odeh;

Все резервуары плагин разделяет на восемь типов, список которых представлен в списке ниже.

1. Насыщенный нефтяной резервуар с изначальной газовой шапкой, без водоносного горизонта.
2. Нефтяной резервуар без водоносного горизонта, с газовой шапкой, формирующейся в процессе добычи.
3. Ненасыщенный нефтяной резервуар без водоносного горизонта.
4. Насыщенный нефтяной резервуар с водоносным горизонтом и изначальной газовой шапкой.
5. Насыщенный нефтяной резервуар с водоносным горизонтом и газовой шапкой, формирующейся в процессе добычи.
6. Ненасыщенный нефтяной резервуар с водоносным горизонтом.
7. Газовый резервуар без водоносного горизонта.
8. Газовый резервуар с водоносным горизонтом.

Тип резервуара определяется входными данными, а именно: флюид, наличие водоносного горизонта, исторические данные, содержащие в себе объем добытой нефти/газа, падение давления, наличие газовой шапки, а также другие необходимые параметры.

Основной процесс модуля «Material Balance» – это «Create Tank» («Создание резервуара»). Для того, чтобы в пользовательском интерфейсе создать объект плагина – резервуар с соответствующими графиками, пользователю необходимо ввести имеющиеся у него данные и при нажатии на кнопку «Run» объекты будут сгенерированы. Далее в программном продукте происходит получение введенных пользователем данных и генерация объектов. На рисунке 5 представлен пользовательский интерфейс плагина для создания объекта Tank (Резервуар).

Рис. 5. Форма создания объекта "Tank"

Как видно из рисунка, на вкладке Settings пользователь может выбрать тип резервуара Oil/Gas и задать для них входные параметры. Помимо вводимых вручную данных для успешного создания объекта необходим так называемый флюид (Black oil model). Флюид – это модель смеси компонентов, находящаяся в коллекторе. Флюид является объектом платформы Petrel и его тестирование не входит в задачи данной работы. Тем не менее, перед проведением тестирования его необходимо создать для использования в дальнейшем.

Для проверки корректности создаваемых объектов и результатов необходимо удостоверится, что все 8 описанных типов резервуаров создаются корректно, для этого подойдет метод функционального тестирования.

2.2. .NET Framework

Petrel разработан с использованием .NET Framework, технологии WPF и языка C#. Это необходимо учитывать при выборе технологии для автоматизации тестирования. .NET Framework – это программная платформа от компании Microsoft. Основной идеей этой платформы является то, что она поддерживает несколько языков, так как код компилируется на общем языке – Common Intermediate Language. Также .Net предоставляет большое разнообразие технологий, которые разработчики используют как для графических приложений, так и для веб-сайтов.

WPF – это часть платформы .NET, предоставляющий возможности для построения графических интерфейсов. В основе WPF лежит графическая технология DirectX. Используя эту технологию, пользователь не фиксирует элементы управления как в WinForms, а использует язык разметки XAML. Помимо этого, WPF имеет еще ряд преимуществ по сравнению с WinForms, например:

- стили и шаблоны;
- анимация;
- поддержка воспроизведения аудио- и видеофайлов;
- модель рисования;
- и др.

Язык программирования C# - это объектно-ориентированный язык, он спроектирован специально для .Net Framework. C# испытал влияние нескольких языков программирования, в особенности C++ и Java.

В среде разработки Microsoft Visual Studio существует возможность подключения дополнительных фреймворков для написания автоматических тестов.

2.3. Фреймворк NUnit

Фреймворк NUnit предназначен для написания и запуска тестов на языках Microsoft .Net. NUnit – это часть парадигмы разработки «Test-Driven-Development» или «Разработка через тестирование».

В NUnit используются атрибуты для обозначения классов и методов. Атрибут [Test Fixture] означает, что данный класс содержит в себе автоматические тесты и по факту это можно назвать тест сьютом (Test Suite). Методы, находящиеся в этом классе, помечаются атрибутом [Test], и они являются тест кейсами (Test Case). Результаты выполненных тестов предоставляются сразу и никаких субъективных человеческих интерпретаций не требуется [11]. Если тест в Test Explorer’e «зеленый», то он пройден, если «красный» - это означает, что тест провален.

Если необходимо выполнить какое-либо действие до или после всех тестовых методов из тест сьюта, то для этого используют следующие атрибуты:

- [TestFixtureSetUp] – для выполнения перед запуском тестов;
- [TestFixtureTearDown] – для выполнения после запуска тестов.

Атрибут [SetUp] говорит о том, что в текущем тестовом наборе метод должен выполнятся после каждого теста, а атрибут [TearDown] говорит о том, что метод запускается уже после каждого теста.

Это лишь основные атрибуты, используемые во фреймфорке NUnit, в следующей главе будут рассмотрены используемые атрибуты непосредственно на примерах для тестируемого продукта.

2.4. Среда тестирования Ocean API

При написании плагинов, а также автоматизированных тестов, предназначенных для платформы Petrel, необходимо использовать Ocean API.

Ocean API предоставляет среду для тестирования, которую можно использовать для запуска автоматизированных тестов. Тесты запускаются при каждой автоматической сборке и после изменений в коде. На данный момент среда ориентирована исключительно на функциональное тестирование и не поддерживает возможность создания тестов с использованием пользовательского интерфейса. Для создания и запуска тестов также необходим фреймворк NUnit для Visual Studio, который также позволяет запускать интеграционные тесты.

2.5. Visual Studio Team System

Для организации работы команды в Schlumberger используется Visual Studio Team System. Его функционал позволяет грамотно скоординировать работу разработчиков, менеджеров и тестировщиков. Visual Studio Team System (VSTS) — это решение от Microsoft, включающее в себя систему управления версиями, сбор данных, построение отчетов, отслеживание статусов и изменений по проекту. Серверная часть Visual Studio Team System используется серверная инфраструктура и хранилище данных Team Foundation Server (TFS), который обеспечивает систему репозиторием и контролем за рабочими элементами. Также серверная часть включается в себя Build Agent – компоненту, которая отвечает за сборки проектов.

В клиентской части для пользователя существует такое понятие как Backlog, который строится в соответствии с методикой управления на

проекте. В Schlumberger команда, работающая над продуктом, рассматриваемым в этой работе, придерживается методологии Agile. Backlog включает в себя следующие основные элементы, которые представлены с описанием в таблице 3.

Таблица 3. Основные элементы Backlog'а

Элемент	Описание
Requirements	Требования, выдвинутые заказчиком
User stories	История пользователя, основанная на описанном requirement'е
Tasks	Конкретная задача, привязанная к одной User story
Bugs	Ошибка, найденная в продукте, которая, как правило, привязывается к user story

Эти элементы называются work items, с помощью них и происходит управление проектом в целом. Все они связываются в иерархию и каждому члену команды понятно, например, к какой user story относится bug. Каждый элемент имеет несколько состояний, которые характеризуют, в какой стадии он находится, например: New, Active, Closed. Также есть поле «Assigned To», для того, чтобы было понятно, кто является ответственным за выполнение этого элемента. На рисунке 6 представлен пример того, как выглядит Backlog в VSTS.

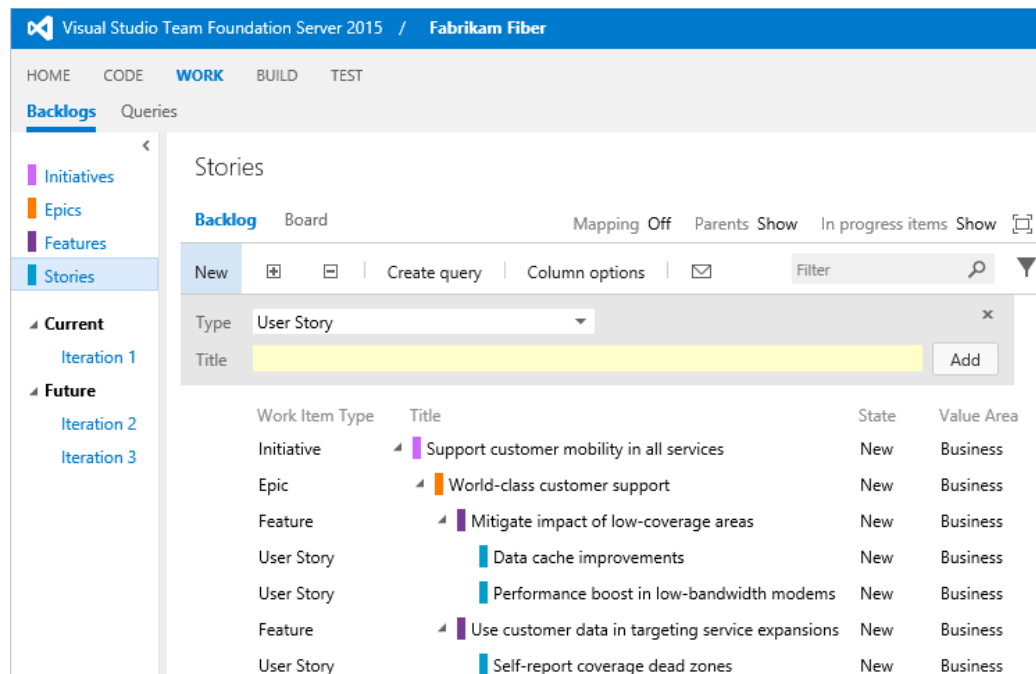

Рис. 6. Пример Backlog'a

Помимо вышеописанного, в VSTS существует раздел Test Plan, в котором инженеры по качеству могут создавать тест-планы и тест кейсы в ручном режиме. Именно в этом разделе будет создан тест план для ручного тестирования плагина.

2.6. Разработка набора тестов и подбор тестовых данных

Перед проведением автоматизации необходимо решить, какие тесты будут необходимы, а также подобрать соответствующие тестовые данные. Последние играют важную роль в проведении тестирования, поскольку от того, что пришло на вход, напрямую влияет результат, который увидит пользователь. После изучения тестируемого объекта, было решено, что для того, чтобы удостоверится в работоспособности функционала, в первую очередь нужно протестировать создание всех возможных типов резервуаров и рассчитанные параметры для них, а далее результирующие графики. Стоит

отметить, что для пользователя графики представлены в виде «картинок», которые отображаются на специальных окнах платформы Petrel, но их построение происходит строго на рассчитанных значениях в результате выполнения определенных функций плагина.

Так как при создании набора тестов было решено опираться на возможные типы резервуаров, описанные в предыдущей главе, то первым будет рассмотрен резервуар ненасыщенный без водоносного горизонта. Для того чтобы он таковым являлся, необходимо чтобы давление во время добычи не упало ниже давления насыщения пластового флюида (Bubble point pressure), и, как видно из названия, отсутствовал водоносный горизонт. Его отсутствие определяется тем, что в процессе добычи не происходит выкачивания воды из резервуара, соответственно при использовании плагина не нужно вводить никаких данных по производству воды. Условно входные данные для теста можно разделить на три части:

- флюид;
- история добычи;
- остальные входные параметры, требуемые плагином.

Флюид является объектом платформы Petrel и используется в работе плагина, поэтому при тестировании необходимо создать такой объект с соответствующими параметрами. Именно у этого объекта задается точка давления насыщения, для ненасыщенного резервуара установим это значение в 80 бар. Поскольку это довольно низкое давление, то будет не сложно подобрать исторические данные, в которых давление не достигнет такого уровня. Прочие параметры флюида представлены ниже:

- минимальное давление – 80 бар;
- максимальное давление – 350 бар;
- температура – 76.85 С;

- эталонное давление – 350 бар;

Необходимые входные параметры для нефтяного коллектора с их диапазонами для проведения расчетов представлены в таблице 4.

Таблица 4. Диапазоны значений входных параметров

Входной параметр	Описание параметра
Давление	200-400 бар
Губина скважины	1500-3000м
Температура	Зависит от глубины и температурного градиента (его среднее значение ~2,94ºC на каждые 100м), а стандартная пластовая температура 38-93ºC.
Начальные запасы нефти	72794500 (это число не должно превышать значение в исторических данных на момент окончания добычи)
Газовая шапка	Равна 0, поскольку резервуар ненасыщенный
Пористость	У коллекторов пористость составляет 0,05-0,3, но у большинства находится в диапазоне 0,1-0,2.
Насыщенность	Среднее значение 0,2
Газовый фактор (GOR)	$GOR = \frac{G_p}{N_p}$, где G_p – суммарная добыча газа, N_p – суммарная добыча нефти
Сжимаемость воды	В режиме истощения пласта среднее значение этого параметра = 4,35* $10^{-5}\ ^{1}/_{бар}$. (Л. П. Дейк)
Сжимаемость породы	Математическая формула расчета сжимаемости: $c = \frac{1}{V}\frac{\partial V}{\partial p}$, где V – объем, p – давление. Л. П. Дейк в книге «Практика инжиниринга

	нефтяных пластов» [19] пишет, что поровая сжимаемость достаточно мала и, как правило, типичные значения лежат в диапазоне 4 - 90 * $10^{-5}\ ^{1}/_{бар}$.

При подборе исторических данных для ненасыщенного нефтяного резервуара без водоносного горизонта необходимо учесть, что давление по окончании добычи не должно упасть ниже давления насыщения, которое было установлено в 80 бар, поэтому минимальное давление устанавливается на 165 бар. Для проверки способности продукта работать с различными типами резервуаров, не имеет смысла брать большой промежуток истории добычи, поэтому было выбрано 10 временных точек, которые представлены в таблице 5.

Таблица 5. Исторические данные

Дата	Кумулятивная добыча нефти (м3)	Кумулятивная добыча газа (м3)	Давление
1/1/2010	0	0	362
2/1/2010	500	16399.5	361
5/1/2010	2000	65598	361
14/1/2010	6500	213193.5	360
10/2/2010	20000	955980	356
2/5/2010	60500	1984339.5	345
1/1/2011	182500	5985817.5	313
1/1/2012	365000	11971635	268
1/1/2013	548000	17973852	229
1/10/2014	730500	23959670	194
1/1/2015	913000	29945488	165

По такому же принципу подобраны данные для тестирования остальных типов коллекторов, в частности, для коллектора с газовой шапкой изначальное давление должно быть уже меньше давления насыщения.

При тестировании вручную инженеру по тестированию необходимо вводить все данные самостоятельно для проверки каждого типа, что занимает достаточно большое количество времени.

Когда тесты на создание резервуара будут пройдены, можно начинать проверку остальных компонентов, а именно графиков. Рассмотрим Energy plot, для пользователя такой график выглядит как на рисунке 7, где каждый цвет обозначает свой показатель, а именно:

- Connate Water Expansion (CWDI);
- Initial Oil Zone Expansion (DDI);
- Initial Gas Cap Expansion (SDI);
- Rock Expansion (PVDI);
- Net Water Influx (WDI);
- Injected Gas Expansion (GIDI);
- Injected Water Expansion (WIDI).

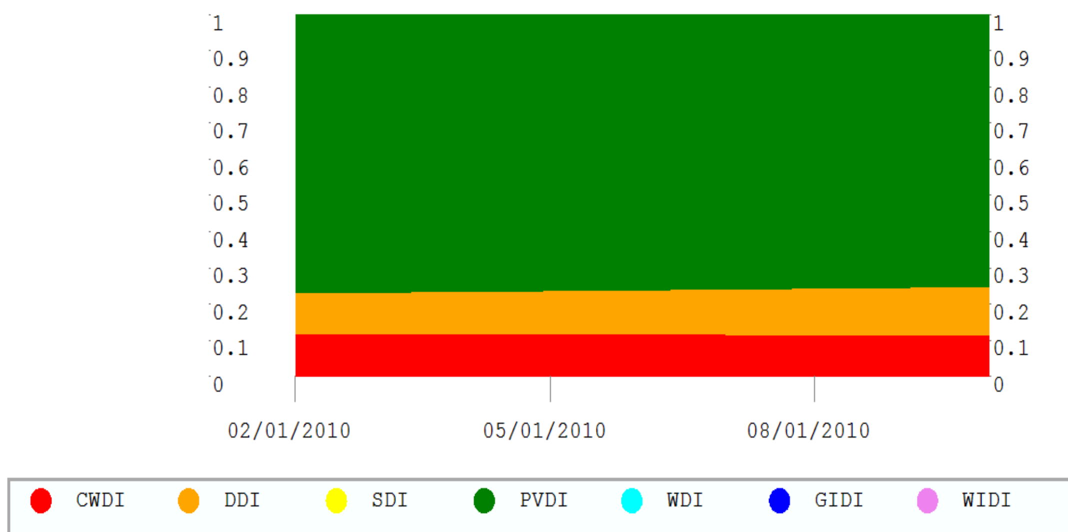

Рис. 7. Energy plot

Присутствие всех показателей на графике одновременно не является обязательным, так как если пользователь работает, например, с резервуаром без водоносного горизонта, то Net Water Influx не будет влиять на вид графика, так как этот параметр равен 0.

Для того чтобы проверить корректность расчета параметров Energy plot, имеет смысл взять такой пример, где за один раз можно проверить максимальное число параметров – это нефтяной или газовый резервуар с водоносным горизонтом.

Следующий элемент, который подлежит тестированию – это график Havlena Odeh, который иллюстрирует добычу нефти. Расчеты, выполненные для его построения, считаются верными, а входные данные корректными, когда все значения на графике лежат на одной прямой под углом 45°. Пример, как выглядит график, представлен на рисунке 9. Для того чтобы удостоверится, что расчеты для построения графика выполнены верно, необходимо проверить координаты этих точек, что будет реализовано в следующем пункте этой главы.

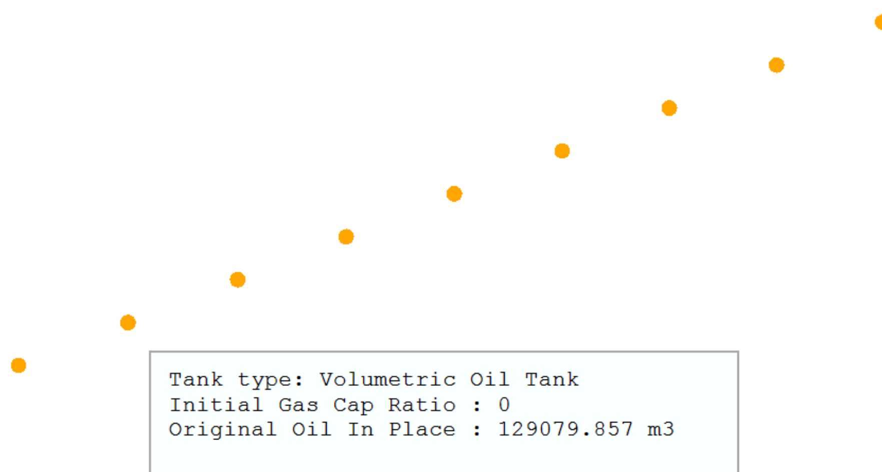

Рис. 8. График Havlena Odeh

После того, как продукт был изучен, был написан набор ручных тестов. В таблице 6 представлен пример тест кейса на русском языке, являющийся аналогом кейса на английском, для ненасыщенного нефтяного резервуара без водоносного горизонта.

На рисунке 9 представлен один из тестовых наборов, разработанных для последующей автоматизации. Тестовый набор содержит 8 ручных тест кейсов для тестирования 8 типов резервуаров.

Outcome	Or...	ID	Title	Configuration	Tester	State
Passed	1	32236	Always undersaturated oil reservoir without water influx	Default configuration ...	Anastasia Deriglazova	Ready
Passed	2	32292	Oil reservoir without water influx and gas cap will form later	Default configuration ...	Anastasia Deriglazova	Ready
Passed	3	30260	Oil reservoir without water influx with initial gas cap	Default configuration ...	Anastasia Deriglazova	Ready
Passed	4	32242	Always undersaturated oil reservoir with water influx	Default configuration ...	Anastasia Deriglazova	Ready
Passed	5	35025	Oil reservoir with water influx without initial gas cap but gas cap will form later	Default configuration ...	Anastasia Deriglazova	Ready
Passed	6	32266	Oil reservoir with water influx with initial gas cap	Default configuration ...	Anastasia Deriglazova	Ready
Passed	7	32248	Gas reservoir without water influx	Default configuration ...	Anastasia Deriglazova	Ready
Passed	8	32294	Gas reservoir with water influx	Default configuration ...	Anastasia Deriglazova	Ready

Рис. 9. Ручные тест кейсы для создания всех типов резервуаров

Таблица 6. Пример ручного тест кейса

Шаг	Действие	Ожидаемый результат
1	Откройте окно плагина	Окно плагина открылось
2	Выберете «Create new» и введите имя «j0»	Имя введено
3	Нажмите на поле «Black oil model» и выберете флюид «j0» из выпадающего списка	Флюид выбран
4	Введите следующие значения в секцию «Reservoir»: •Давление: 362 бар •Глубина: -2100 м •Температура: 76,85ºC •Соотношение нефти и газа: 32,8$м^3/м^3$ •Сжимаемость воды: 3,7733e-05 1/бар	Значения введены успешно
5	Выберете «Specified» для переменной «Rock Compressibility»	«Specified» выбрано для переменной «Rock Compressibility»
6	Введите 2,21893e-05 1/бар в поле «Rock Compressibility»	Значение введено успешно
7	Введите 72794500 в поле «Original oil in place»	Значение введено успешно
8	Введите 0 в поле «Initial Gas Cap Ratio»	Значение введено успешно
9	Введите 0.2 в поле «Porosity»	Значение введено успешно
10	Введите 0.2 в поле «Water Saturation»	Значение введено успешно
11	На вкладку «History» выберете исторические данные из выпадающего	Исторические данные отобразились в таблице

	списка	
12	Нажмите кнопку «Run»	Резервуар был создан. Графики Havlena Odeh и Energy plot созданы.
13	Отобразите график Havlena Odeh	Грфик представлен в виде прямо линии по углом 45°

2.7. Автоматизация тестов

В предыдущем параграфе было описано, какие тестовые случаи необходимы для продукта и также были выбраны тестовые данные для проверки всех возможных типов резервуаров, так как за их создание отвечают разные классы и методы. Для того, чтобы сократить временные затраты на тестирование программного продукта, были разработаны автоматизированные тесты. Их можно запускать в среде разработки Visual Studio.

Для написания автоматических тестов был создан проект модульного теста (рис. 10), который был подключен к основному проекту плагина «Material Balance» в Visual Studio (рис. 11).

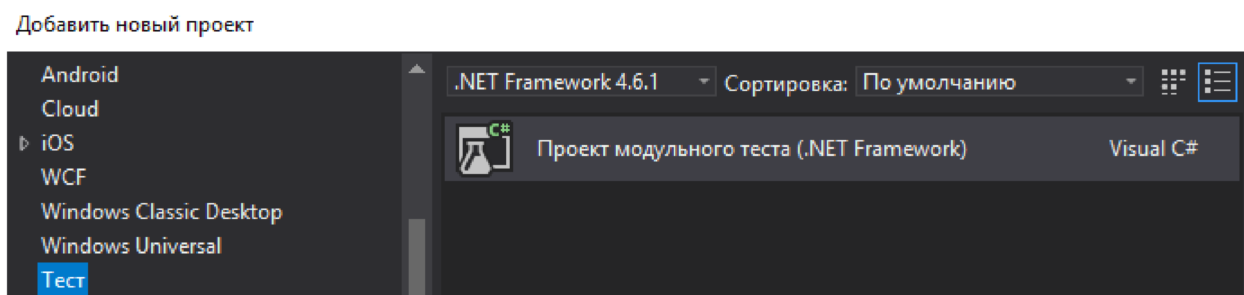

Рис. 10. Создание проекта для автоматических тестов

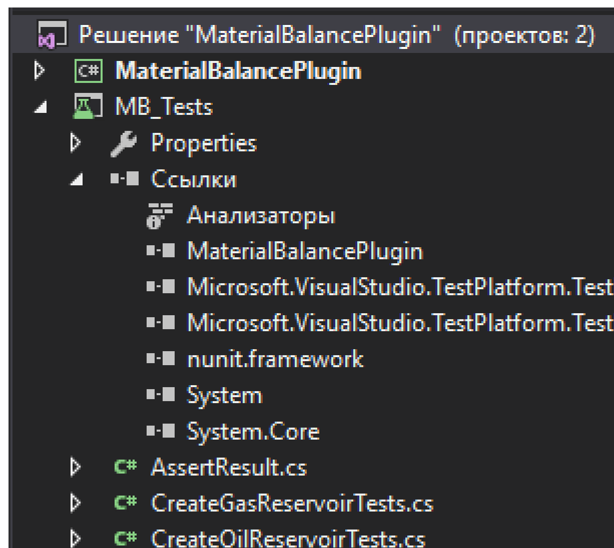

Рис. 11. Дерево проектов в решении «Material Balance Plugin»

Для того чтобы пометить классы и методы как тестовые был использован NUnit Framework, предназначение которого было описано в параграфе 2.3 настоящей главы.

Когда пользователь заполняет форму плагина «Создание резервуара», при выборе типа резервуара Oil и установке параметра Aquifer в none, внесенные данные записываются в соответствующие поля класса плагина *Tank.cs.* Далее создается объект и производится расчет неизвестных параметров. Для того чтобы тестировщику не приходилось имитировать работу пользователя и заполнять интерфейс вручную, были созданы файлы в формате .csv с входными данными для каждого типа объекта. Для считывания и инициализации этих данных реализован класс *InputData.cs*, в методы которого можно передается необходимый файл. Эти методы являются вспомогательными и не помечаются атрибутом [Test], так как они не участвуют в тестовом покрытии кода плагина. Метод *initializeOilTank()* реализован для инициализации входных значений нефтяного коллектора. Метод возвращает объект типа Tank, т.е. объект резервуара, параметры которого необходимо протестировать. Для проверки обычного газового

резервуара были разработан аналогичный метод – *initializeGasTank ()* (см. приложение 4).

При разработке функциональных тестов для плагина «Material Balance» было решено начать с создания теста для ненасыщенного нефтяного резервуара без водоносного горизонта. Как было описано выше, в плагине имеется класс *Tank.cs*, который отвечает за инициализацию основных и расчет неизвестных параметров нефтяного и газового коллекторов и наследуется от классов *OilProperties.cs* и *GasProperties.cs.* Свойства нефтяного резервуара этого класса, представлены в таблице 7.

Таблица 7. Свойства класса OilProperties.cs

Поле	Значение
Pressure	Давление
DatumElevation	Глубина
Temperature	Температура
InitialGOR	Соотношение газа и нефти
WaterCompressibility	Сжимаемость воды
RockCompressibility	Сжимаемость породы
OriginalOilInPlace	Начальные запасы нефти
InitialGasCap	Газовая шапка
Porosity	Пористость
WaterSaturation	Насыщенность

Для тестирования объекта резервуара, реализация которого представлена в классе *Tank.cs* проекта «MaterialBalancePlugin», были созданы два класса *CreateOilReservoirTests.cs* и *CreateGasReservoirTest.cs,* для того, чтобы при выполнении тестов, инженеру-тестировщику было легче идентифицировать, где произошел сбой.

Архитектура автоматических тестов, для проверки расчетных значений тестируемого класса *Tank.cs* соответствует структуре ручных тест кейсов, а именно:

- предусловия;
- шаги выполнения;
- сравнение с ожидаемым результатом;

Методы в классе *CreateOilReservoirTests.cs* реализованы согласно паттерну написания тестов AAA, о котором было сказано в пункте 1.5.1 настоящей работы. В этом классе реализованы методы, с помощью которых происходит процесс тестирования различных параметров нефтяного резервуара. Так, для того, чтобы удостоверится, что тип «Ненасыщенный нефтяной резервуар без водоносного горизонта» рассчитывается верно, а именно, коллектор является ненасыщенным, необходимо проверить, что рассчитанное значение газовой шапки (Gas cap) равно нулю. Для этого реализован метод *GasCap_0returned(),* в котором проверяется значение параметра CalculatedGasCap. На вход методу подается файл, содержащий входные параметры для инициализации резервуара. На рисунке 12 представлен пример этого метода.

```
[TestMethod]
public void GasCap_0returned()
{
    //Arrange
    string oilPropertiesInput = InputData.ReadFileOil("UndersaturatedOilNoAquifer.csv");

    //Act
    Tank tankj0 = CreateOilReservoirTests.initializeOilTank(oilPropertiesInput);
    tankj0.CreateOilReservoir(tankj0.Pressure, tankj0.DatumElevation, tankj0.Temperature,
        tankj0.InitialGOR, tankj0.WaterCompressibility, tankj0.RockCompressibility,
        tankj0.OriginalOilInPlace, tankj0.InitialGasCap, tankj0.Porosity, tankj0.WaterSaturation);

    //Assert
    Assert.AreEqual(0, tankj0.CalculatedGasCap);
}
```

Рис. 12. Метод для проверки значения параметра газовой шапки.

Таким образом, процесс тестирования выполняется поэтапно.

1. Считывается файл с тестовыми данными.
2. Создается объект типа Tank для дальнейшего тестирования.
3. Считанные данные записываются в объект.
4. Вызывается тестируемая функция.
5. Результат выполнения функции сравнивается с ожидаемым значением.

Таим образом, с помощью методов *initializeOilTank()* и *initializeGasTank()* эмулируется ввод данных в пользовательский интерфейс. В таком случае инженеру по тестированию не требуется вводить большое количество данных для проверки работоспособности всех объектов, и, следовательно, экономится большое количество времени.

В каждом из классов *CreateOilReservoirTests.cs* и *CreateGasReservoirTests.cs* реализованы методы для проверки всех рассчитываемых параметров, которые необходимы при использовании метода материального баланса. В каждом из этих методов вызываются соответствующие функции плагина, отвечающие за создание объектов с тестовыми данными и расчет свойств. Для проверки корректности этих свойств используется методы класса *Assert.cs,* например, *AreEqual()*, в который передается ожидаемое и фактическое значения, а также значение дельты, если допустимо расхождение в этих значениях.

Такие тестовые случаи были реализованы для проверки всех параметров, значений которых рассчитываются в плагине для получения результатов, необходимых пользователю программного продукта.

В плагине «Material Balance» создание каждого графика происходит в классе с соответствующим названием. Расчет значений для графиков происходит после создания доменного объекта. Для Energy plot график представлен в виде прямоугольника, который состоит из параметров, описанных в предыдущем параграфе, следовательно, автоматизированные

тесты необходимо реализовать таким образом, чтобы можно было проверить каждый из этих параметров. Процесс тестирования Energy plot заключается в том, чтобы все параметры резервуара, влияющие на график, имели определенное значение. График Havlena Odeh строится по другому принципу, так как он состоит из точек, выстроенных в линию, то в этом случае необходим корректный расчет координат, на которых следует находится каждой точке.

Рассмотрим график Energy plot, который строится для нефтяного резервуара. Для работы с таким графиком плагин использует класс *EnergyPlot.cs.* Для его тестирования был создан класс *TestEnergyPlot.cs.* Значения для описываемого графика рассчитываются только для резервуаров, тип которых Oil. Было реализовано четыре метода, с именами, отражающих «что тестируется» и «у чего тестируется», например, *CheckDriveIndices_InitialGasCapNoAquifer().*

Перед проверкой любых значений, в функции создается объект резервуара с заранее определенными тестовыми параметрами. А после выполнения части кода плагина, где выполнен расчет необходимого критерия, с помощью функции *Assert.AreEqual()* рассчитывается разница между полученным и ожидаемым значениями (пример представлен в приложении 3). Ниже на рисунке 13 представлен результаты тестов, проверяющих расчеты для Energy plot, и поскольку такой график присутствует у шести из восьми возможных типов резервуаров, то тестовых методов тоже шесть.

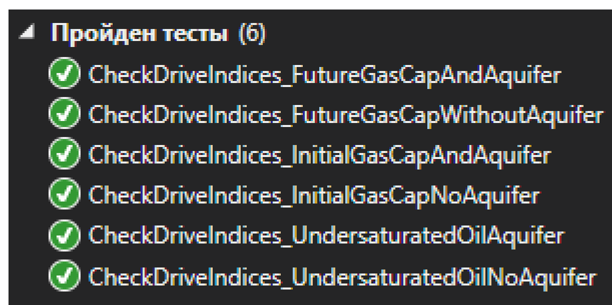

Рис. 13. Результат выполнения автоматических тестов для Energy plot

Для тестирования класса построения графика HavlenaOdeh.cs был реализован класс HavlenaOdehTests.cs. Процесс тестирования функций построения этого вида графика заключается в проверке координат точек, из которых состоит график. В свойствах xValuesTest и yValuesTest, являющимися массивами реализованного класса, записаны эталонные значения для координат построения графика. Эталонные значения сравниваются с фактическими, используя класс CollectionAssert фреймворка NUnit.

Результаты всех запущенных тестов можно увидеть в Test Explorer'е, в том виде, как это представлено на рисунке 13. Тесты, которые были выполнены успешно, отмечены зеленой иконкой, а те, которые провалились – красной. Все вышеописанное было реализовано в рамках автоматизации тестирования. Теперь инженеру по тестированию не придется каждый раз вводить одни и те же значения для проверки функциональности и сравнивать все параметры вручную. Также, после выполнения тестов, можно сразу легко понять в какой функции произошла ошибка.

2.8. Полученная выгода после внедрения автоматизации

В начале написания магистерской диссертации одной из проблем было то, что при выполнении некоторых видов тестирования, например, регрессионного, тратится большое количество времени. Для этого была проделана работа, описанная в предыдущих параграфах, а именно автоматизация тестирования, так как в этом случае инженерам по тестированию не придется выполнять вручную все тесты для проверки работоспособности плагина.

Ниже будет представлен сравнительный анализ времени, которое тратилось на написание и прохождение ручных и автоматических тестов. В. М. Гребенюк в своей работе о целесообразности внедрения автоматизации [18] приводит формулы для расчета временных затрат на автоматизированное тестирование и тестирование при помощи ручных тестов.

По формуле 1 можно провести расчет временных инвестиций на автоматизацию тестирования за выделенный период времени. За период времени будет взят 1 год. Описание всех переменных формулы приведены в таблице 1, в последней колонке которой представлено значение, которое будет использовано в расчетах в рамках рассматриваемой задачи.

$$I_p = I_0 + C_0 + \sum_{n=1}^{k}(C_e + C_a + C_m) \tag{1}$$

Таблица 8. Описание параметров формулы (1)

Параметр	Описание	Используемое значение (количество часов)
I_0	Время, потраченное на поиск подходящего ПО и обучение сотрудников	15
C_0	Среднее время, потраченное на написание тестов	50
k	Количество циклов тестирования	12
C_e	Время, потраченное на подготовку и выполнение одного теста, умноженное на общее количество тестов	0,5
C_a	Время, потраченное на анализ результатов	0,5
C_m	Время, потраченное на поддержание автоматизированных тестов в актуальном состоянии	0

$I_p = 15 + 50 + \sum_{n=1}^{12}(0{,}5 + 0{,}5 + 0) = 77$ ~ 10 рабочих дней.

Формула 2 иллюстрирует, как можно посчитать временные затраты на ручное тестирование, а в таблице 2 представлены описания и значения переменных формулы.

$$G_p = G_0 + \sum_{n=1}^{k}(G_e + G_a + G_m) \qquad (2)$$

Таблица 9. Описание параметров формулы (2)

Параметр	Описание	Используемое значение (количество часов)
G_0	Временные затраты на написание ручных тестов	60
k	Количество циклов тестирования	12
G_e	Время, потраченное на выполнение ручных тестов за один цикл	12
G_a	Время, потраченное на анализ результатов ручного тестирования	0,25
G_m	Время, потраченное на поддержание ручных тестов в актуальном состоянии	0

$I_p = 80 + \sum_{n=1}^{12}(12 + 0{,}25 + 0) = 207 \sim 26$ рабочих дней.

Таким образом, сравнив два полученных результата можно увидеть, что автоматизация оправдана по временным затратам. Возможно, на начальных этапах ожидалась чуть большая выгода. Тем не менее, экономия присутствует, ведь прохождение автотестов занимает несколько минут и может быть запущено в любой момент, в отличие от прохождения ручных тестов.

Заключение

В ходе написания выпускной квалификационной работы были изучены виды и методы тестирования программных продуктов, а также инструменты, применяемые для написания ручных и автоматизированных тестов. Одной из поставленных задач в данной работе была автоматизация тестов для тестирования плагина, предназначенного для платформы Petrel. Для того чтобы выполнить это задачу, в первую очередь был изучен сам продукт и его процессы. В ходе изучения продукта выяснилось, как можно систематизировать создаваемые объекты для написания компактного набора тестов, а также определены наиболее проблемные и ответственные этапы и процедуры процесса тестирования. Например, при создании резервуара создаются графики, которые необходимы конечному пользователю для понимания результатов расчета.

Была проведена автоматизация тестирования плагина «Material Balance», вследствие чего временные затраты на проведение тестирования уменьшились в сравнении с тестированием при использовании ручных тестов. Таким образом, все поставленные задачи выполнены, а цель выпускной квалификационной работы достигнута. В дальнейшем планируется провести автоматизацию тестирования пользовательского интерфейса, поскольку на выполнение этого вида тестирования также требуются большие временные затраты, которые можно легко снизить используя автоматизацию.

Список литературы

1. Guide to the software engineering body of knowledge [Электронный ресурс]. – URL: http://www4.ncsu.edu/~tjmenzie/ cs510/pdf/SWEBOK-v3.pdf. (15.01.19)
2. Ron Patton, Software Testing. – Sams Publishing, 2005. – 408 с.
3. Куликов С., Тестирование программного обеспечения. Базовый курс. – Минск: Четыре четверти , 2017. — 298 с.
4. ISTQB Glossary [Электронный ресурс]. – URL: https://glossary.istqb.org/search/. (25.02.19)
5. Уроки BDD: Ручное тестирование [Электронный ресурс]. – URL: http://software-testing.ru/library/testing/general-testing/2798-bdd-101ma-nual-testing. (05.03.19)
6. Автоматизация тестирования [Электронный ресурс]. – URL: http://aplana.ru/services/testing/avtomatizaciya-testirovanija. (07.03.19)
7. DSL Engineering [Электронный ресурс]. – URL: http://voelter.de/dslbook/markusvoelter-dslengineering-1.0.pdf. (28.03.19)
8. Roy Osherove, The art of Unit Testing with Examples in .NET. – Manning Publications 2009. – 320с.
9. Meszaros G., xUnit Test Patterns. Refactoring Test Patterns. – Boston: Addison-Wesley, 2007. – 948 с.
10. Симан М., Внедрение зависимостей в .NET. – СПб: Питер, 2013. – 464с.
11. NUnit Documentation [Электронный ресурс]. – URL: http://nunit.org/documentation/. (19.04.19)
12. Gherkin notation [Электронный ресурс]. – URL: http://docs.behat.org/en/v2.5/guides/1.gherkin.html. (02.04.19)
13. Бейзер Б., Black-box testing. – СПб: Питер, 2004. – 320с.

14. Spillner A., Linz T., Software Testing Foundations: A Study Guide for the Certified Tester Exam. – Rocky Nook, 2014. – 289c.
15. Murnane T., Reed R., On the effectiveness of mutation analysis as a black box testing technique. Материалы 13-ой Австралийской конференции по разработке программного обеспечения. – Австралия, 2001.
16. Канер С., Тестирование программного обеспечения. Фундаментальные концепции менеджмента бизнес-приложений. – ДиаСофт, 2001. – 544 с.
17. Савин Р., Тестирование Дот Ком. – М.: Дело, 2007. – 311с.
18. Оценка целесообразности внедрения автоматизированного тестирования [Электронный ресурс]. – URL: https://cyberleninka.ru/article/v/otsenka-tselesoobraznosti-vnedreniya-avtomatizirovannogo-testirovaniya. (01.06.19)
19. Дейк Л.П., Практика инжиниринга нефтяных пластов М.: Премиум Инжиниринг", 2009. – 570 с.

Приложение 1. Классы для тестирования создания нефтяного резервуара

```
    [TestFixture]
    public class CreateOilReservoirTests
    {

        [TestCase]
        public void GasCap_0returned()
        {
            //Arrange
            string oilPropertiesInput =
InputData.ReadFile("UndersaturatedOilNoAquifer.csv");
            Tank tank = InputData.initializeOilTank(oilPropertiesInput);

            //Act
            tank.CreateOilReservoir();
            tank.CalculateGasCap();

            //Assert
            Assert.AreEqual(0, tank.CalculatedGasCap);
        }

        [TestCase]
        public void checkOilFormationVolumeFactor()
        {
            //Arrange
            string oilPropertiesInput =
InputData.ReadFile("UndersaturatedOilNoAquifer.csv");
            Tank tank = InputData.initializeOilTank(oilPropertiesInput);

            //Act
            tank.CreateOilReservoir();
            tank.CalculateOilFormationVolumeFactor();

            //Assert
            Assert.AreEqual(0.000862, tank.OilFormationVolumeFactor);

        }

        [TestCase]
        public void OOIP_129079returned()
        {
            //Arrange
            string oilPropertiesInput =
InputData.ReadFile("OilInitialGasCapNoAquifer.csv");
            Tank tank = InputData.initializeOilTank(oilPropertiesInput);


            //Act
            tank.CreateOilReservoir();
            tank.CalculateOOIP();

            //Assert
            Assert.AreEqual(129079.857, tank.CalculatedGasCap, 0.2);

        }
```

```
        [TestCase]
        public void GasCap_076returned()
        {
            //Arrange
            string oilPropertiesInput =
InputData.ReadFile("OilInitialGasCapNoAquifer.csv");

            //Act
            Tank tank = InputData.initializeOilTank(oilPropertiesInput);
            tank.CreateOilReservoir();
            tank.CalculateGasCap();

            //Assert
            Assert.AreEqual(0.76, tank.CalculatedGasCap);
        }
}
```

Приложение 2. Класс для тестирования создания газового резервуара

```
[TestFixture]
public class CreateGasReservoirTests
{
    [TestCase]
    public void OGIP_65325660returned()
    {
        //Arrange
        string gasPropertiesInput = InputData.ReadFile("GasNoAquifer.csv");
        Tank tank = InputData.initializeGasTank(gasPropertiesInput);

        //Act
        tank.CreateGasReservoir();
        tank.CalculateOGIP();

        //Assert
        Assert.AreEqual(65325660.857, tank.CalculatedGasCap);

    }


    [TestCase]
    public void checkGasFormationVolumeFactor()
    {
        //Arrange
        string gasPropertiesInput = InputData.ReadFile("GasNoAquifer.csv");
        Tank tank = InputData.initializeGasTank(gasPropertiesInput);

        //Act
        tank.CreateGasReservoir();
        tank.CalculateGasFormationVolumeFactor();

        //Assert
        Assert.AreEqual(0.000862, tank.GasFormationVolumeFactor);

    }


    [TestCase]
    public void checkTotalWaterFlowforGas()
    {
        //Arrange
        string gasPropertiesInput = InputData.ReadFile("GasWithAquifer.csv");
        Tank tank = InputData.initializeGasTank(gasPropertiesInput);

        //Act
        tank.CreateGasReservoir();
        tank.CalculateGasFormationVolumeFactor();

        //Assert
        Assert.AreEqual(35.596732, tank.OilFormationVolumeFactor);

    }
```

```
[TestCase]
public void WaterFlowforGas_0return()
{
    //Arrange
    string gasPropertiesInput = InputData.ReadFile("GasNoAquifer.csv");
    Tank tank = InputData.initializeGasTank(gasPropertiesInput);

    //Act
    tank.CreateGasReservoir();
    tank.CalculateWaterFlow();

    //Assert
    Assert.AreEqual(0.0, tank.GasFormationVolumeFactor);

}
   }
```

Приложение 3. Класс для тестирования графика Energy plot

```
    [TestFixture]
    public class EnergyPlotTests
    {
        [TestCase]
        public void CheckDriveIndices_InitialGasCapNoAquifer()
        {

            string oilPropertiesInput = InputData.ReadFile("InitialGasCapNoAquifer.csv");
            Tank tank = InputData.initializeOilTank(oilPropertiesInput);

            tank.CreateOilReservoir();

            var energyPlot = new EnergyPlot();
            energyPlot.calculateDriveIndices(tank);

            Assert.AreEqual(energyPlot.ConnateWaterExpansion, 0.0, 10e-7); // CWDI
            Assert.AreEqual(energyPlot.InitialOilZoneExpansion, 0.49064377, 10e-7); //
DDI
            Assert.AreEqual(energyPlot.InitialGasCapExpansion, 0.01530002, 10e-7); // SDI
            Assert.AreEqual(energyPlot.RockExpansion, 0.03650961, 10e-6); // PVDI
            Assert.AreEqual(energyPlot.NetWaterInflux, 0.32381839, 10e-6); // WDI
            Assert.AreEqual(energyPlot.InjectedGasExpansion, 0.0, 10e-7); // GIDI
            Assert.AreEqual(energyPlot.InjectedWaterExpansion, 0.0, 10e-7); // WIDI

        }

        [TestCase]
        public void CheckDriveIndices_FutureGasCapWithoutAquifer()
        {

            string oilPropertiesInput =
InputData.ReadFile("FutureGasCapWithoutAquifer.csv");
            Tank tank = InputData.initializeOilTank(oilPropertiesInput);

            tank.CreateOilReservoir();

            var energyPlot = new EnergyPlot();
            energyPlot.calculateDriveIndices(tank);

            Assert.AreEqual(energyPlot.ConnateWaterExpansion, 0.0, 10e-7); // CWDI
            Assert.AreEqual(energyPlot.InitialOilZoneExpansion, 0.52064387, 10e-7); //
DDI
            Assert.AreEqual(energyPlot.InitialGasCapExpansion, 0.00025400, 10e-7); // SDI
            Assert.AreEqual(energyPlot.RockExpansion, 0.0657981, 10e-6); // PVDI
            Assert.AreEqual(energyPlot.NetWaterInflux, 0.4238887, 10e-6); // WDI
            Assert.AreEqual(energyPlot.InjectedGasExpansion, 0.0, 10e-7); // GIDI
            Assert.AreEqual(energyPlot.InjectedWaterExpansion, 0.0, 10e-7); // WIDI


        }

        [TestCase]
        public void CheckDriveIndices_UndersaturatedOilNoAquifer()
        {
```

```
            string oilPropertiesInput =
InputData.ReadFile("UndersaturatedOilNoAquifer.csv");
            Tank tank = InputData.initializeOilTank(oilPropertiesInput);

            tank.CreateOilReservoir();

            var energyPlot = new EnergyPlot();
            energyPlot.calculateDriveIndices(tank);

            Assert.AreEqual(energyPlot.ConnateWaterExpansion, 0.0, 10e-7); // CWDI
            Assert.AreEqual(energyPlot.InitialOilZoneExpansion, 0.8904307, 10e-7); // DDI
            Assert.AreEqual(energyPlot.InitialGasCapExpansion, 0.0, 10e-7); // SDI
            Assert.AreEqual(energyPlot.RockExpansion, 0.08250961, 10e-6); // PVDI
            Assert.AreEqual(energyPlot.NetWaterInflux, 0.0, 10e-6); // WDI
            Assert.AreEqual(energyPlot.InjectedGasExpansion, 0.0, 10e-7); // GIDI
            Assert.AreEqual(energyPlot.InjectedWaterExpansion, 0.0, 10e-7); // WIDI


        }

}
```

Приложение 4. Класс для тестирования графика Havlena Odeh

```
    [TestFixture]
    public class HavlenaOdehTests
    {

        [TestCase]
        public void HavlenaOdehGasNoAquifer()
        {

            string gasPropertiesInput = InputData.ReadFile("GasNoAquifer.csv");
            Tank tank = InputData.initializeGasTank(gasPropertiesInput);

            tank.CreateGasReservoir();
            HavlenaOdehPlot.xValuesOfHO(tank);
            HavlenaOdehPlot.yValuesOfHO(tank);

            double[] xValuesTest = { 0.0355, 0.1220, 0.1097, 0.1096, 0.1085 };
            double[] yValuesTest = { 1.392747E+07, 3.751092E+07, 3.753401E+07,
3.758495E+07, 3.760460E+07 };

            for (int index = 0; index < 5; ++index)
            {
                Assert.AreEqual(xValuesTest[index], HavlenaOdehPlot.xValues[index], 1e-
3);
                Assert.AreEqual(yValuesTest[index], HavlenaOdehPlot.yValues[index], 1e-
6);
            }
        }


        [TestCase]
        public void HavlenaOdehOilWithAquifer()
        {

            string oilPropertiesInput =
InputData.ReadFile("UndersaturatedOilWithAquifer.csv");
            Tank tank = InputData.initializeOilTank(oilPropertiesInput);

            tank.CreateOilReservoir();
            HavlenaOdehPlot.xValuesOfHO(tank);
            HavlenaOdehPlot.yValuesOfHO(tank);

            double[] xValuesTest = { 0.0320, 0.1180, 0.1077, 0.1086, 0.1084 };
            double[] yValuesTest = { 1.85322E+07, 1.998652E+07, 2.758801E+07,
3.685525E+07, 3.799689E+07 };

            for (int index = 0; index < 5; ++index)
            {
                Assert.AreEqual(xValuesTest[index], HavlenaOdehPlot.xValues[index], 1e-
3);
                Assert.AreEqual(yValuesTest[index], HavlenaOdehPlot.yValues[index], 1e-
6);
            }
        }
```

```
        [TestCase]
        public void HavlenaOdehOilNoAquifer()
        {

            string oilPropertiesInput =
InputData.ReadFile("UndersaturatedOilNoAquifer.csv");
            Tank tank = InputData.initializeOilTank(oilPropertiesInput);

            tank.CreateOilReservoir();
            HavlenaOdehPlot.xValuesOfHO(tank);
            HavlenaOdehPlot.yValuesOfHO(tank);

            double[] xValuesTest = { 0.0400, 0.1100, 0.1095, 0.1086, 0.1085 };
            double[] yValuesTest = { 1.392747E+07, 3.751092E+07, 3.753401E+07,
3.758495E+07, 3.760460E+07 };

            for (int index = 0; index < 5; ++index)
            {
                Assert.AreEqual(xValuesTest[index], HavlenaOdehPlot.xValues[index], 1e-
3);
                Assert.AreEqual(yValuesTest[index], HavlenaOdehPlot.yValues[index], 1e-
6);
            }

        }
```

Приложение 5. Класс для инициализации тестовых параметров

```
public static class InputData
    {

        public static Tank initializeOilTank(string properties)
        {
            Tank tank = new Tank();
            string[] splitedProperties = properties.Split(';');
            tank.Pressure = Convert.ToDouble(splitedProperties[1],
NumberFormatInfo.InvariantInfo);
            tank.DatumElevation = Convert.ToDouble(splitedProperties[2],
NumberFormatInfo.InvariantInfo);
            tank.Temperature = Convert.ToDouble(splitedProperties[3],
NumberFormatInfo.InvariantInfo);
            tank.InitialGOR = Convert.ToDouble(splitedProperties[4],
NumberFormatInfo.InvariantInfo);
            tank.WaterCompressibility = Convert.ToDouble(splitedProperties[5],
NumberFormatInfo.InvariantInfo);
            tank.RockCompressibility = Convert.ToDouble(splitedProperties[6],
NumberFormatInfo.InvariantInfo);
            tank.OriginalOilInPlace = Convert.ToDouble(splitedProperties[7],
NumberFormatInfo.InvariantInfo);
            tank.InitialGasCap = Convert.ToDouble(splitedProperties[8],
NumberFormatInfo.InvariantInfo);
            tank.Porosity = Convert.ToDouble(splitedProperties[9],
NumberFormatInfo.InvariantInfo);
            tank.WaterSaturation = Convert.ToDouble(splitedProperties[10],
NumberFormatInfo.InvariantInfo);
            tank.OriginalOilInPlace = Convert.ToDouble(properties[11],
NumberFormatInfo.InvariantInfo);
            return tank;
        }


        public static Tank initializeGasTank(string properties)
        {
            Tank tank = new Tank();
            string[] splitedProperties = properties.Split(';');
            tank.Pressure = Convert.ToDouble(splitedProperties[1],
NumberFormatInfo.InvariantInfo);
            tank.DatumElevation = Convert.ToDouble(splitedProperties[2],
NumberFormatInfo.InvariantInfo);
            tank.Temperature = Convert.ToDouble(splitedProperties[3],
NumberFormatInfo.InvariantInfo);
            tank.OriginalGasInPlace = Convert.ToDouble(splitedProperties[4],
NumberFormatInfo.InvariantInfo);
            return tank;
        }


        public static string ReadFile(string filename)
        {
            string line;
            string result = "";
```

```
        using (StreamReader sr = new StreamReader(filename))
        {
            while ((line = sr.ReadLine()) != null)
            {
                result += line;
            }
        }
        return result;
    }
   }
```