**5 ПРОГРАММА И МЕТОДИКА ИСПЫТАНИЙ**

Большинство создаваемых в данное время Rails-приложений разрабатывается с применением TDD (test-driven development) методологии. Данная методология разработки программного обеспечения предполагает повторение коротких циклов разработки: в первую очередь пишется тест, который покроет изменение, а затем пишется код, который пройдет тест. Ближе к концу разработки проводится рефакторинг нового кода для приведения его к соответствующим стандартам. Такой подход является полной противоположностью разработке, при которой сначала разрабатывается программное обеспечение, а затем описываются тестовые ситуации.

Тест – это операция, с помощью которой определяется работоспособен ли данный код. При проверке написанного кода программист выполняет тестирование функционала вручную. В данной ситуации тестирование состоит из двух стадий: стимулирование кода и проверка выполнения его функционала. Автоматическое тестирование выполняется по-другому: вместо программиста стимулированием кода и проверкой его выполнения занимается компьютер, на дисплее которого отображается результат прохождения теста: код работоспособен или код неработоспособен. Таким образом происходит «инверсия ответственности»: от реализации тестов и их детальности зависит, будет ли код соответствовать техническому заданию. Методика разработки через тестирование заключается главным образом в реализации автоматических тестов.

Разработка через тестирование требует от разработчика написания автоматизированных модульных тестов, определяющих требования к коду непосредственно перед написанием самого кода. Тест содержит проверки условий, которые могут либо выполняться, либо нет. Когда условия выполняются, говорят, что тест пройден. Прохождение теста подтверждает поведение, предполагаемое программистом. Разработчики часто пользуются библиотеками для тестирования для создания и автоматизации запуска наборов тестов. На практике модульные тесты покрывают критические и нетривиальные участки кода. Это может быть код, который подвержен частым изменениям, код, от работы которого зависит работоспособность большого количества другого кода, или код с большим количеством зависимостей.

Среда разработки должна быстро реагировать на небольшие модификации кода. Архитектура программы должна базироваться на использовании множества сильно связанных компонентов, которые слабо соединены друг с другом, благодаря чему тестирование кода упрощается.

TDD не только предполагает проверку исправности работы, но и влияет на дизайн программы. Опираясь на тесты, разработчики могут подробнее представить, какая функциональность необходима пользователю. Таким образом, детали интерфейса появляются задолго до окончательной реализации решения.

К тестам применяются те же требования стандартов кодирования, что и к основному коду.

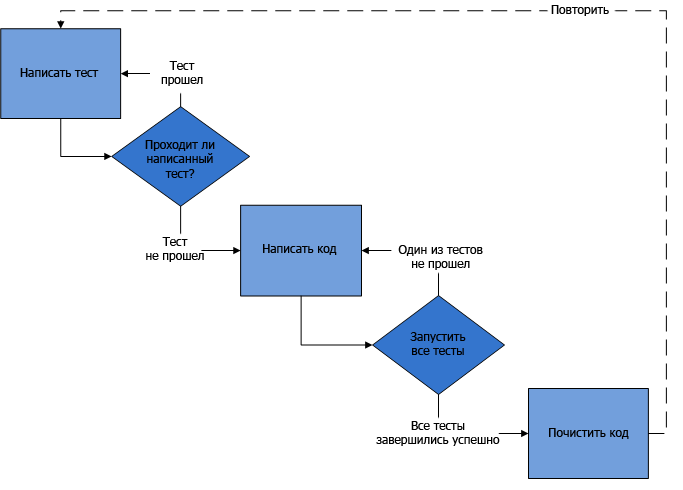


Рисунок 5.1 – Разработка с использованием TDD

Современные проекты предъявляют высокие требования к покрытию автоматическими тестами. Написание тестов является одним из главных требований при разработке и написании кода программного обеспечения в наше время. Все чаще мы слышим такие аббревиатуры, как TDD (Test Driven Development) и BDD (Behaviour Driven Development) и многие строго следуют этим подходам в разработке. BDD это одна из разновидностей TDD, и именно такая парадигма используется в системе RSpec.

RSpec – это фреймворк для тестирования написанный на языке программирования Ruby и предоставляющий специальный DSL (domain-specific programming language) для написания тестов – спецификаций. RSpec – инструмент для BDD. Спецификация (spec) – это обычно отдельный файл, который содержит описание какой-нибудь части программы, в контексте Rails, это может быть описание целого контроллера, модели, шаблона, партиала, хелпера и т.д. Файлы спецификаций принято хранить в поддиректории spec проекта, а имена файлов должны заканчиваться на \_spec.rb.

Часто случаются ситуации, когда добавление нового функционала ведет к изменению поведения старого, и в такой ситуации разработчик может об этом даже не узнать. Такая ситуация имеет свое название – регрессия.

Регрессионное тестирование – это виды тестирования программного обеспечения, позволяющие обнаружить ошибки в уже протестированном ранее функционале. Регрессионное тестирование выполняется не для того, чтобы убедиться в отсутствии ошибок в имеющемся функционале, а для исправления регрессионных ошибок (ошибки, появляющиеся не при написании программы, а при добавлении в исходный код нового функционала или исправление ошибок, что и стало причиной возникновения новых ошибок в уже протестированной программе). Цикл регрессионного тестирования представлен на рисунке 5.2.

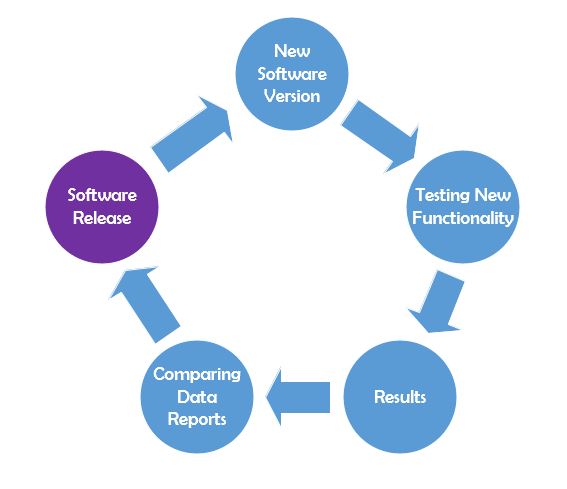


Рисунок 5.2 – Цикл регрессионного тестирования

Методы регрессионного тестирования включают в себя многократное выполнение предыдущих тестов и проверку того, что добавление нового функционала не вызвало создание новых регрессионных ошибок.

Регрессионное тестирование включает в себя три этапа:

- верификация устранения нового дефекта;

- верификация того, что дефект, который был проверен и исправлен ранее, не воспроизводится вновь;

- верификация того, что функциональность приложения не нарушилась после исправления дефектов и внесения нового кода.

Выделяют несколько видов регрессионных тестов:

- верификационные тесты (проводятся для проверки исправления обнаруженной ранее ошибки);

- тестирование верификации версии (проверка работоспособности основной функциональности программы).

При написании исходного кода веб-приложения тестирование проводилась в три этапа:

- тестирование отдельно каждого сервиса в процессе написания программного кода;

- тестирование взаимодействия нескольких сервисов между собой;

- полное тестирование программы после окончания процесса написания программного кода.

Данные этапы являются важными и связанными между собой, ни один из них невозможно исключить. Например, без модульного тестирования, при анализе работы программы в целом, будет происходить достаточное количество сбоев, выявить которые может оказаться достаточно сложным, в то время как при анализе работы одного модуля неисправность оказывается достаточно очевидной. И обратный случай, работоспособность каждого компонента в отдельности не гарантирует корректное поведение всей программы в целом.

При написании веб-приложения проводилось юнит-тестирование. Отдельно необходимо упомянуть сквозное тестирование.

**5.1 Юнит-тестирование**

Для тестирования сервисов серверной части использовалось юнит-тестирование. Под юнит-тестированием подразумевается процесс, позволяющий проверить на корректность отдельные модули исходного кода. Основная цель юнит-тестирования заключается в том, чтобы писать тесты для каждой функции или метода и проверять как позитивные, так и негативные сценарии. Это позволяет быстро проверить, не привело ли добавление нового функционала к регрессии.

**5.2 Сквозное тестирование (end-to-end)**

Сквозные тесты выполняют тестирование системы в целом, эмулируя реальную пользовательскую среду. Данные тесты позволяют протестировать полный цикл работы какого-либо сценария, начиная от пользовательского интерфейса и до отправки запроса на сервер. Сквозные тесты проверяют взаимодействие сервисов между собой. В интернете это тесты, запущенные в браузере, имитирующие щелчки мышью и нажатия клавиш.

Для сквозного тестирования в проектах, созданных с помощью фреймворка Ruby on Rails, использовался фреймворк Capybara. Данный фреймворк имеет понятную и объемную документацию. На официальном сайте можно найти всю информацию по фреймворку, основные его плюсы и возможности.