Министерство образования и науки Российской Федерации

Федеральное государственное автономное образовательное учреждение
высшего профессионального образования
«Московский физико-технический институт
(государственный университет)»

Факультет управления и прикладной математики

Кафедра теоретической и прикладной информатики

ОПТИМИЗАЦИЯ ПРОЦЕССА ТЕСТИРОВАНИЯ ПО С ИСПОЛЬЗОВАНИЕМ ЭВРИСТИЧЕСКОГО АНАЛИЗА ТЕСТОВОГО ПОКРЫТИЯ КОДА

Выпускная квалификационная работа
(магистерская работа)

Направление подготовки: 03.04.01 Прикладные математика и физика

Выполнил:
Смирнов Илья Алексеевич
Научный руководитель:
к.ф-м.н. Коротаев К.С.
Рецензент:
старший программист Орлов Д.Л.

Москва - 2017

Содержание

0. Введение

Тестирование программного обеспечения - процесс анализа программного средства и сопутствующей документации с целью выявления дефектов и повышения качества продукта. Этап тестирования является неотъемлемой частью цикла жизни программного обеспечения.[1]

В момент зарождения индустрии разработки это был предельно формализованный процесс, больше похожий на отладку и математическое доказательство правильности кода. С развитием индустрии разработки программного обеспечения и усложнением технологического процесса, появлением комплексных методов решения задач, роль тестирования возросла. Вместо одной из финальных стадия создания проекта тестирование стало применяться на протяжении всего цикла разработки. В дальнейшем были созданы гибкие методики тестирования, разнообразные оптимизации и была углублена интеграция с процессом разработки. В данный момент в индустрии имеется огромный интерес к тестированию, как таковому, так и к методам улучшения качества программ в целом.[2]

В прикладном понимании тестирование определяется как процесс определения качества программного обеспечения в соответствии со стандартом *ГОСТ Р ИСО/МЭК 25010-2015* как степень удовлетворения системой заявленных и подразумеваемых потребностей различных заинтересованных сторон. [20]

Модель качества продукта по данному стандарту обозначается в виде восьми критериев:

- Функциональная пригодность
- Уровень производительности
- Совместимость
- Удобство пользования

- Надежность
- Защищенность
- Сопровождаемость
- Переносимость

Также можно определить тестирование как процесс, имеющий в качестве входных параметров тестируемый объект и набор *кейсов*. Кейс - это пара, определяющая соответствие входных параметров уже тестируемого объекта и ожидаемого от него результата.

0.1 Классификация тестирования

Существует множество признаков, по которым данный процесс можно разделить на классификации. Рассмотрим их подробнее.

Во-первых, это разделение по объекту тестирования. В данном случае упор делается на проверку соответствия заявленных функций действительности. К данному типу тестирования относится тестирование производительности, безопасности, функциональное и юзабилити-тестирование. Обычно, данные проверки производятся отдельным персоналом, не имеющим отношения к непосредственному написанию кода, и производятся уже после формального завершения разработки части продукта.[7]

Во-вторых - по степени знания системы. Классическое разделение случаев белого-черного ящика. В первом случае производится проверка внутренней структуры продукта путем анализа логики работы программы. Во втором - производится проверка поведения продукта с точки зрения внешнего мира, при котором подразумевается отсутствие знаний о его устройстве.[8]

В-третьих, по степени изолированности системы. В данной классификации работа с продуктом ведется с точки зрения разбиения его на логические

компоненты. Тестирование идет от малого к большему - сначала производится проверка изолированных частей, в том числе с использованием имитации других модулей (mocking) - модульное или *юнит-тестирование*. Далее идет интеграционное тестирование - протестированные по отдельности компоненты объединяются в группы и анализируется их совместимость и надежность. Системное тестирование является финальным этапом, в ходе которого проверяется соответствие системы исходным требованиям.[3]

Также существует множество других критериев и видов тестирования, порожденных высокой скоростью развития индустрии и постоянно изменяющимися представлениями о современном и оптимальном процессе разработки.

0.2 Тестирование как этап разработки

Тестирование - единственный способ предотвращения негативных последствий некорректной работы продукта до того, как он попадет к конечному потребителю. Более того, внутренние затраты на проверку качества программ почти всегда покрывают потенциальные риски. Однако, зачастую, в силу ограниченности ресурсов, разработчик вынужден балансировать между количеством и качеством - которое напрямую зависит от этапа тестирования - реализованных задач.

Повышенное уделение внимания тестированию может, в свою очередь, очень сильно тормозить процесс непосредственного написания кода. В случае неоптимального использования тестовых методик, пусть даже и работающих полностью верно, затраты на анализ внесенных изменений имеют тенденцию превосходить затраты на их имплементацию в разы. Интеграция автоматических систем запуска тестов и анализа кода позволяет не допускать попадания в кодовую базу некорректных

коммитов, однако приводит к ситуациям, когда разработчик вынужден ждать часами, пока несколько строк кода попадут куда нужно. Не говоря уже о всевозможных неполадках, которые могут возникнуть в процессе самого тестирования. Не смотря на эти проблемы, отказ от тестирования является куда большим риском. Именно по-этому прилагаются серьезные усилия по оптимизации и стандартизации тестирования. Одной из таких оптимизаций и посвящена данная работа.

1. Подробнее о юнит-тестировании

В исследовании основной упор будет сделан на оптимизацию юнит-тестирования. Идея метода состоит в том, чтобы покрыть все нетривиальные участки кода тестами, описывающими всевозможные кейсы. Обычно, написанием тестов занимается непосредственно разработчик, работающий над тестируемой частью.

1.1 Преимущества

1.1.1. Упрощение процесса разработки

Уже существующие на момент внесения изменений в кодовую базу тесты фиксируют корректное поведение программы, что позволяет избежать регрессов - появления ошибок в ранее работавших корректно частях. Добавление тестов, посвященных конкретным нетривиальным исправлениям, также позволяет быстрее понять, какого типа проблемы были вызваны неудачными изменениями.

1.1.2. Документирование

Каждый тест, будучи набором входных данных и соответствующих им результатов, определяет правила, по которым работает тестируемый код. При написании тестов поощряется стремление к простоте и возможности быть прочитанным “сверху-вниз”.

1.1.3. Стандартизация интерфейсов

Тестирование подобным методом позволяет гарантировать совместимость модулей друг с другом, что упрощает последующий процесс интеграционного тестирования.[11]

1.1.4. Рефакторинг

С помощью аналитических результатов тестов, таких как покрытие или времена работы отдельных функций, разработчик может обнаруживать “мертвый” или неоптимальный код.

1.1.5. Разработка через тестирование (TDD)

Заключается в инвертировании процесса разработки модуля. Сначала, по требованиям, происходит создание тестов. В дальнейшем, они выступают “каркасом” для настоящего кода, который постепенно проходит все тесты. В дальнейшем проводится рефакторинг написанного кода, защищенного тестами от регрессов. Необходимо указать, что важной частью разработки через тестирование является сепарирование тестов друг от друга - при написании нового теста в первую очередь проверяется, что он не проходит. Это гарантирует покрытие каждым тестом конкретно своего случая.[6]

1.2. Недостатки

К сожалению, юнит-тестирование не всегда дает только лишь преимущества. К недостаткам и сложностям данного метода можно отнести следующее.

1.2.1. Громоздкость

Допустим, тестируемый код состоит из последовательного принятия решений в зависимости от значений булевых переменных. Комбинаторным путем легко можно посчитать количество тестов, требующихся для его покрытия. А в случае комплексных проверок результатов, а также учитывая необходимость имитации работы сторонних модулей, объем тестирующего кода может превосходить тестируемый разы, а иногда и в десятки раз.

1.2.2. Зависимость от культуры разработки

Строгие правила, которым необходимо следовать при использовании данной технологии, не всегда являются либо понятными разработчику, либо приемлемыми с точки зрения компании. Требуется отслеживание данных обо всех тестах, использование системы контроля версий, а также

отсутствие хаотического изменения требований к модулю. В противном случае преимущества данного метода не будут проявляться, а само тестирование лишь замедлит разработку.

1.2.3. Низкая полезность в отсутствие других этапов тестирования

Не смотря на то, что, с точки зрения требований, модуль будет работать корректно, при отсутствии более высокоуровневых проверок тестируемый код может оказаться бесполезным. Любой проект, для которого актуально тестирование, состоит из многих частей, и именно результат их совместной работы интересует конечного потребителя, а не корректность работы из по-отдельности.

1.2.4. Устаревание и неверная работа объектов-заглушек

Одним из принципов юнит-тестирования является сосредоточение внимания конкретного теста на конкретной части кода. Для этого разработчик создает примитивные объекты-заглушки, подменяя ими вызовы других модулей. Фактически, идет дублирование кода, что приводит к ситуации, когда разработчики вносят изменения в реальный код, но забывают про изменение работы имитации. Последствиями этого являются ошибки интеграции и то, что код завязывается на неверный функционал.

1.3. Вывод

Так как любой процесс разработки проходит в условиях ограниченности ресурсов, описанные выше проблемы имеют тенденцию проявляться в любых более-менее крупных проектах. Они неизбежны, однако отказ от тестирования приводит к куда более катастрофическим последствиям.

2. Покрытие кода

В данном исследовании нас интересует такая характеристика, как *покрытие (coverage) кода*. Покрытие определяется как множество затронутых набором тестов элементов кода. Анализ покрытия позволяет:

- Обнаружить части программы, не затронутые тестами
- Создать дополнительные тесты, чтобы повысить покрытие
- Получить количественные характеристики покрытия, оценивая таким образом качество тестов
- Обнаружить пересечения тестируемых случаев, что позволяет избавиться от лишних тестов
- Обнаружить избыточные условия
- Выявить неадекватность требований к программе

Важно помнить, что покрытие определяет качество набора тестов, но не тестируемого кода.

2.1. Метрики

Существует множество метрик, различным способом характеризующих уровень покрытия. Опишем их, от более простых к более комплексным.

2.1.1. Построчное покрытие (Statement Coverage)

Для обеспечения полного покрытия программного кода на данном уровне необходимо, чтобы в результате выполнения тестов каждый оператор был выполнен хотя бы один раз. Наиболее примитивный вариант оценки - полное покрытие означает, что из всех возможных логических комбинаций тесты смогли предоставить как минимум одну. Описанный ниже пример будет иметь полное построчное покрытие, но будет пропущена проверка существенной части логики, а именно ошибка при передаче *False* в качестве параметра функции.

```
def foo(condition: bool):
    if condition:
        result = True
    return result

assertTrue(foo(True))
```

2.1.2. Покрытие ветвей (Decision Coverage)

Данный метод ориентирован на покрытие всех возможных логических путей исполнения тестируемой программы. Учитываются также не описанные варианты, как, например, отсутствие *else* в коде выше. Для полного его покрытия потребуется уже два теста (пусть код и работает некорректно):

```
assertTrue(foo(True))
assertRaises(UnboundLocalError, foo, False)
```

Данный случай, однако, не предоставляет верного анализа логических условий. Для демонстрации этого, модифицируем пример:

```
def foo(condition_one: bool, condition_two: bool):
    if condition_one and (condition_two or important_code_call()):
        return True
    else:
        return False

assertTrue(foo(True, True))
assertFalse( foo(False, False))
```

Покрытие будет полным, однако *important_code_call* так и не будет вызвана.

2.1.3. Покрытие по условиям (Condition Coverage)

Данный метод требует для полного покрытия предоставление всех возможных значений параметров для логического условия. Однако принятие всех возможных значений самим условием не требуется.

2.1.4. Покрытие по веткам и условиям (C/D Coverage)

Сочетание двух предыдущих методов. Но и это не позволяет полностью протестировать правильность логической функции.[13]

2.1.5. Покрытие по всем условиям (Multiple Condition Coverage)

По настоящему “полное” покрытие, требует предоставления всех возможных наборов значений элементов логического условия. Данный метод считается избыточным и проверяется редко. Зависит от структуры условий - для разных условий с одинаковым количеством операторов может понадобиться разное количество тестов.[13]

2.1.6. MC/DC (Modified Condition/Decision Coverage)

Данный метод разработан компанией Boeing и используется для уменьшения количества тестов при тестировании логических условий. Полное покрытие достигается следующим образом:

- Логическое условие должно принимать все возможные значения
- Каждая компонента условия должна принимать все возможные значения хотя бы раз
- Должно быть показано независимое влияние каждой компоненты на значение при фиксации остальных компонент

Не смотря на то, что количество тестов для получения полного покрытия по этой метрике остается большим, оно все-равно значительно меньше, чем использование всех возможных проверок.[13]

3. Проблема длительности тестового цикла

3.1. Описание проблемы

В погоне за полным покрытием, а также в соответствии с требованиями модульного тестирования, разработчики производят огромное количество тестов. А если учитывать то, что в реальности проявляются описанные ранее проблемы, это число растет экспоненциально. В итоге даже в средних проектах, зачастую, разработчику приходится часами ждать проверки единственной правки кода, чтобы продолжить над ним работу. Проявление ошибок в самих тестах, изменение требований, даун-тайм тестирующих машин - все это приводит к торможению процесса. Подобные примеры убеждают некоторых разработчиков вообще отказываться от тестов, а менеджеров - закладываться на меньшую эффективность команды в процессе планирования.

3.2. Варианты решения

Есть два очевидных пути решения данной проблемы - это подход со стороны оптимизации кода и со стороны оптимизации тестов.

3.2.1. Работа с кодовой базой

Рефакторинг, в ходе которого достигается большая обособленность отдельных компонент. Разбиение больших частей на малые, избавление от бесполезного кода и относящихся к нему тестов - все это существенно ускоряет и упрощает этап юнит-тестирования. Однако данный подход не является универсальным. Более того, чем серьезнее проблемы, связанные со сложностью кода, тем труднее и затратнее проводить рефакторинг. В некоторых ситуациях полная переработка с нуля с учетом ошибок и опыта, полученного в ходе работы над предыдущим поколением продукта, будет даже более выгодной, так как увеличивает возможности для дальнейшего развития проекта. К сожалению, данный способ является наиболее

затратным и часто неприемлемым, особенно если разработчик имеет дело с широко распространенным продуктом, имеющим одновременно несколько версий, используемых конечным потребителем.[4]

3.2.2. Оптимизация тестов

Даже в наиболее запущенных случаях существует возможность раздельного тестирования. При внесении изменений в одну часть продукта разработчик предполагает, что благодаря системе интерфейсов и разделению кода, а также опираясь на доказанную предыдущей тестовой итерацией валидность кода, он может с высокой вероятностью протестировать их валидность запустив лишь соответствующую им часть тестов. Данный подход существенно ускоряет разработку, однако вероятность валидной проверки кода теперь опирается не только на качество тестов, но и на знание конкретным человеком набора тестов, требуемых к запуску. Выигрыш времени также варьируется от того, в какой части были внесены изменения - в случае правки редко используемого периферийного метода, для проверки может быть достаточно менее десяти тестов, с учетом только что написанных. Однако изменения в сильно связном коде могут потребовать запуска большинства тестов для подтверждения валидности.

3.3. Причина выбора пути оптимизации

В случае работы с крупным проектом, в котором задействовано множество персонала, серьезное вмешательство в код связано с огромными рисками. Команда разработки, фактически, останавливает работу над новым функционалом и не привносит существенного улучшения производительности - а это именно то, что требуется продукту, чтобы продолжать жить. Даже если менеджмент готов пойти на подобный риск, персонал, зачастую, оказывается не готов к подобным изменениям. При

рефакторинге проявляются скрытые проблемы и противоречия в требованиях, к которым клиенты уже привыкли как к данности не ожидают их переработки. В итоге старые проблемы вынужденно переходят в новый код, забирая с собой, в том числе, и громоздкие тесты.

Конечно же, описанные выше проблемы ни в коем случае не ставят крест на идее рефакторинга и улучшении кодовой базы. Они лишь призваны показать, что небольшой группе разработчиков не под силу решить проблему оптимизации тестирования путем работы с самим кодом. Именно поэтому была выбрана стратегия, подразумевающая полное отсутствие взаимодействия с кодовой базой. Будет произведена оценка существующего набора тестов, какие части кода они затрагивают, а в дальнейшем, без участия человека, будет запускаться оптимальное их подмножество. Непосредственно часть, затрагивающая проект будет сведена к минимуму на финальном этапе встраивания оптимизации в систему контроля внесения изменений в код.

4. Автоматизация процесса тестирования

Процесс тестирования программного обеспечения, при котором основные функции и шаги теста, такие как запуск, инициализация, выполнение, анализ и выдача результата, производятся автоматически с помощью инструментов для автоматизированного тестирования. В свою очередь, инструмент для автоматизированного тестирования — это программное обеспечение, посредством которого осуществляется создание, отладка, выполнение и анализ результатов прогона тест-скриптов (Test Scripts — это наборы инструкций для автоматической проверки определенной части программного обеспечения). Рассмотрим популярный инструментарий travis-ci. Проект непрерывной интеграции в нем является конфигурацией для сервера-исполнителя. По требованию системы создается задача на тестирование, которая помещается в очередь. При обработке задачи сервер применяет относящуюся к ней конфигурацию, после чего запускает тестовые скрипты и, в зависимости от результата и настроек системы, производит различные действия. Например, провал сборки при наличии ошибок в тестах и т.п.

Такие системы внедряются для того, чтобы утилизировать тесты непосредственно в процессе разработки. Рассмотрим в качестве примера популярный сервис github, предоставляющий интерфейс над VCS git. Пользователь может настроить свой репозиторий таким образом, чтобы при запросе на объединение ветвей (pull request) автоматически запускались тесты, чтобы удостовериться, что ошибки не попадут в основную ветвь, а также что общий процент покрытия кода не снизился.[14]

5. Исследуемый метод оптимизации

5.1. Постановка задачи

На вход дается результат работы утилиты анализа покрытия в формате xml, и, опционально, внесенные изменения. Необходимо разработать алгоритм, который позволит максимально проверить код за отведенное ему время, при этом результат работы алгоритма должен быть оптимизирован. Информация о тестах и их соответствии коду хранится отдельно в виде метаданных о текущем состоянии кодовой базы. Метаданные должны итеративно накапливаться и попадать на вход при следующем запуске утилиты. Применение оптимизации должно позволить проводить полный запуск тестов с меньшей частотой и для большего количества изменений за раз.

5.2. Абстрактный метод

Для начала, распишем все возможные варианты работы алгоритма оптимизации в зависимости от входных данных.

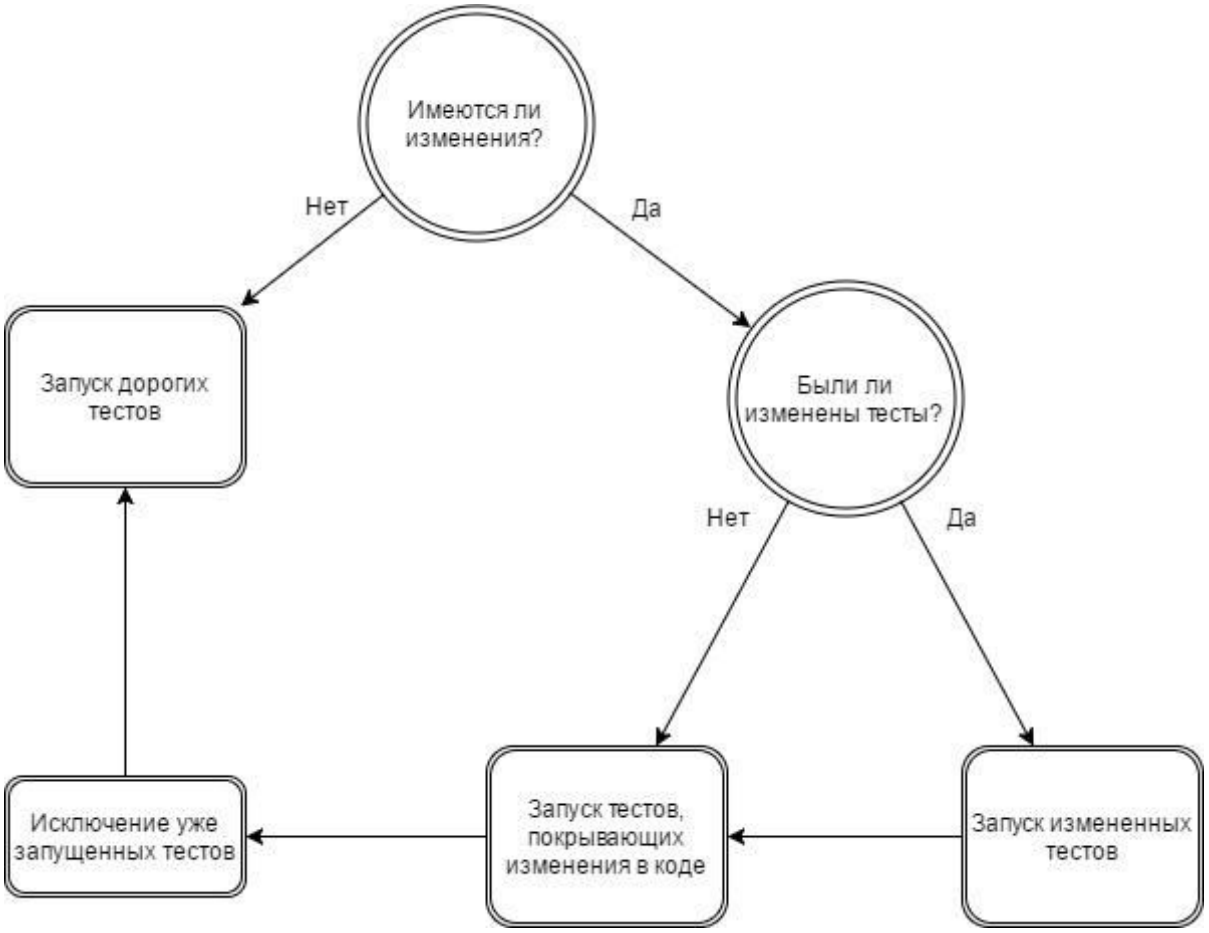
Что может произойти с кодом с точки зрения алгоритма?

Во-первых, код и тесты могли остаться без изменений. В таком случае необходимо вычислить и запустить набор наиболее оптимальных тестов, способных покрыть за отведенное время наибольшее количество строк кода.

Во-вторых, могли быть внесены изменения только в код. В таком случае необходимо в первую очередь протестировать их, для чего требуется определить, какими тестами, вероятно, покрывается данный участок. Добавим их в обязательные для запуска, после чего выберем из оставшихся наиболее оптимальные.

В-третьих, могли быть внесены изменения в сами тесты. Запустим измененные тесты, чтобы получить новую статистику покрытия,

исключим их из рассмотрения, после чего переходим к проверке, описанной выше.



Из описанной схемы видно, что необходимо конкретизировать два алгоритма - выбор “дорогих” тестов и выбор тестов, покрывающих изменения. Необходимо также уточнить некоторые прикладные моменты.

5.2.1. Провал теста

Тесты создаются для того, чтобы знать, что пошло не так в программе. Естественно, они будут проваливаться. В таком случае результаты покрытия данного теста учитываться в будущем не должны.

5.2.2. Типы изменений кода

Существует три типа изменений - модификация строк, удаление строк и добавление строк, и на каждые тип алгоритм должен реагировать по-своему. Данные случаи будут рассмотрены позднее.

5.2.3. Типы изменений тестов

В данном случае все намного проще. В случае удаления теста мы также удаляем данные о нем. Случаи модификации и добавления тестов не отличаются для алгоритма между собой - в обоих вариантах необходимо их запустить.

5.3. Алгоритм выбора “дорогих” тестов

Данный алгоритм работает только с тестами и информацией об их покрытии и времени работы. Изменения не учитываются.

5.3.1. Постановка задачи

Дан набор тестов $\{t_i, c_i\}$, а также T - время, отведенное на тестирование. Необходимо выбрать набор тестов таким образом, чтобы их суммарное покрытие $\cup c_i$ было максимальным, а общее время исполнения было меньше, чем T.

5.3.2. Решение

В первую очередь исключим тесты, время работы которых больше, чем T. К сожалению, насколько бы тест ни был эффективен, мы не можем запустить его, если он не укладывается в базовые требования по времени. Введем функцию веса теста - $m(t, c) = c/t$. Отсортируем тесты по убыванию значения соответствующей им функции. Теперь необходимо избавиться от пересечения покрытий.

Будем последовательно брать наиболее эффективные тесты и отсекать их части покрытия от всех менее эффективных. В итоге каждой части кода будет соответствовать тот тест, который проходит ее наиболее быстро. Визуально этот способ можно продемонстрировать примером ниже. Как мы видим, первый тест, будучи наиболее быстрым, вытеснил собой тесты 2 и 3.

```
#include <stdio.h>                                   123 --> 1
int main()                                                                  123 --> 1
{
   double number;                                                           123 --> 1
   printf("Enter a number: ");                                              123 --> 1
   scanf("%lf", &number);                                                   123 --> 1
   if (number <= 0.0)                                                       123 --> 1
   {
      if (number == 0.0)                                                    12  --> 1
         printf("You entered 0.");                                          1   --> 1
      else
         printf("You entered a negative number.");                          2   --> 2
   }
   else
      printf("You entered a positive number.");                             3   -->
3
   return 0;                                                                123 --> 1
}
```

После данных манипуляций нам потребуется обновленный критерий выбора. Формально задача теперь записывается так.

$$T, \{t_i, c_i\}_1^N \to \{k_1, \ ..., \ k_n\} \ : \ max \sum_{\{k_1, \ ..., \ k_n\}} \bar{c}_i, \quad \sum_{\{k_1, \ ..., \ k_n\}} t_i \le T \ ,$$

$$\text{где } \bar{c}_i : \bar{c}_1 = c_1, \ \bar{c}_k = c_k \bigcap \bigcap_1^{k-1} c_i$$

Данная задача является примером классической “Задачи о рюкзаке” (Knapsack Problem) и решается методом динамического программирования за $O(N \times T)$. В данном случае ценой для предметов будут служить уникальные покрытия, а весом - время исполнения.

5.3.3. Метод динамического программирования

Разберем подробнее решение задачи о рюкзаке.[22] Пусть $A(k, s)$ есть максимальная стоимость предметов, которых можно уложить в рюкзак

вместимости s, если можно использовать только первые k предметов, то есть $\{n_1, ..., n_k\}$, назовем этот набор допустимых предметов для $A(k, s)$.

$A(k, 0) = 0$

$A(0, s) = 0$

Найдем $A(k, s)$. Возможны 2 варианта:

1. Если предмет k не попал в рюкзак. Тогда $A(k, s)$ равно максимальной стоимости рюкзака с такой же вместимостью и набором допустимых предметов $\{n_1, ..., n_{k-1}\}$, то есть $A(k, s) = A(k-1, s)$
2. Если k попал в рюкзак. Тогда $A(k, s)$ равно максимальной стоимости рюкзака, где вес s уменьшаем на вес $k-$ого предмета и набор допустимых предметов $\{n_1, ..., n_{k-1}\}$ плюс стоимость k, то есть $A(k-1, s-w_k) + p_k$

То есть: $A(k, s) = max(A(k-1, s), A(k-1, s-w_k) + p_k)$

Стоимость искомого набора равна $A(N, W)$, так как нужно найти максимальную стоимость рюкзака, где все предметы допустимы и вместимость рюкзака W.

Восстановим набор предметов, входящих в рюкзак. Будем определять, входит ли n_i предмет в искомый набор. Начинаем с элемента $A(i, W)$, где $i = N$ $w = W$,. Для этого сравниваем $A(i, w)$ со следующими значениями:

1. Максимальная стоимость рюкзака с такой же вместимостью и набором допустимых предметов $\{n_1, ..., n_{i-1}\}$, то есть $A(i-1, w)$
2. Максимальная стоимость рюкзака с вместимостью на w_i меньше и набором допустимых предметов $\{n_1, ..., n_{i-1}\}$ плюс стоимость p_i, то есть $A(i-1, w-w_i) + p_i$

Заметим, что при построении A мы выбирали максимум из этих значений и записывали в $A(i,w)$. Тогда будем сравнивать $A(i,w)$ с $A(i-1,w)$, если равны, тогда n_i не входит в искомый набор, иначе входит.

5.3.4. Реализация метода динамического программирования

Будем считать, что все веса нормализованы.

```
def knapsack_solution(N, W, w):
    A = [[0 for _ in range(W)] for _ in range(N)]
    for k in range(N):
        for s in range(W):
            if s >= w[k]:
                A[k][s] = max(A[k-1][s], A[k-1][s-w[k]]+p[k])
            else:
                A[k][s] = A[k-1][s]
    answer = list()
    def get_best(k, s):
        if A[k][s] == 0:
            return
        if A[k-1][s] == A[k][s]:
            get_best(k-1, s)
        else:
            get_best(k-1, s - w[k])
            answer.append(k)
    get_best(N, W)
    return answer
```

5.4. Алгоритм для работы с изменениями

Теперь необходимо рассмотреть алгоритм, который был бы применим для работы с изменениями в коде. В первую очередь заметим, что никакие оптимизации не будут иметь смысла, если непосредственно изменения не будут протестированы. Однако мы вынуждены работать с ограничением по времени. Таким образом, не всегда получится полностью проверить все возможные ошибки, особенно с учетом того, что тесты не являются истиной в последней инстанции.

Алгоритм будет простым - определим, к каким областям покрытия относятся изменения, после чего, используя описанный выше алгоритм, выберем столько затронутых тестов, сколько получится запустить, не превышая данный лимит времени.

5.4.1. Определение затронутых тестов

Для каждой строки существует три варианта изменений:

1. Модификация. В данном случае необходимо запустить тесты, которые ранее покрывали данную строку.
2. Удаление. Является подклассом модификации. Также запускаем тесты, которые ранее покрывали данную строку.
3. Добавление новых строк. Наиболее сложный случай. Так как мы работаем с абстрактными данными о покрытии, мы не имеем никакой информации о синтаксическом дереве кода. В данном случае предлагается запускать тесты, покрывавшие строки, номера которых заняли новые, а также строку перед внесенными изменениями (с точки зрения покрытия). Такое решение не гарантирует, что новый код будет покрыт. Простой пример - код выше и ниже новых строк - отдельные функции, никак с ним не связанные. С другой стороны, согласно культуре разработки, разработчик обязан добавить тесты, покрывающие новый код. Так что основным источником риска в данном случае является разработчик, но не утилита.

6. Имплементация

В данной части будет описан процесс разработки непосредственно оптимизационной утилиты, а также возникшие в ходе него проблемы и их решения.

6.1. Требования к имплементации

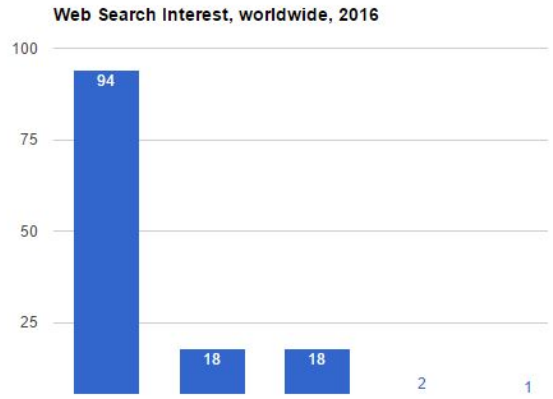
Для реализации алгоритма оптимизации требуется наличие следующего функционала:

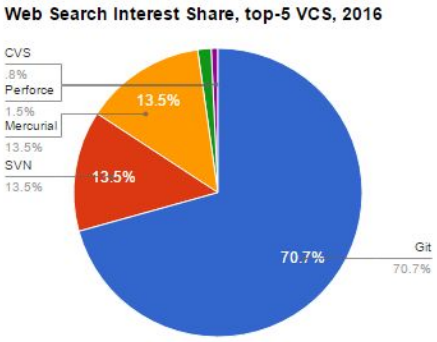
1. Работа с изменениями в кодовой базе. Возможность определять, какой тип модификации был произведен над кодом.
2. Использование распространенной и выгодной утилиты, предоставляющей результаты покрытия
3. Интерфейс для выбранной утилиты
4. Хранение данных о тестах и покрытии
5. Сопоставление покрытия, кода и изменений
6. Вычисление разницы между областями покрытия
7. Интерфейс для работы с тестирующей системой и интеграции
8. Инициализация и поддержание актуальности своих данных
9. Возможность запуска на различных платформах и различных окружения

6.2. Работа с изменениями в кодовой базе

Так как одной из основных задач является упрощение процесса разработки, предполагается, что утилита должна отслеживать, какие части кода были изменены и каким образом. Согласно Eclipse Community Survey, в котором респонденты в том числе, указывали используемую систему контроля версий, в 2014 году git обошел svn по статистике использования. Данные выводы можно также подтвердить статистикой Google по поисковым запросам о VSC за 2016 год, в которой git занимает 70,7%. git

diff предоставляет весь функционал, необходимый для имплементации утилиты. Более того, сама система может помочь в решении некоторых дополнительных задач, в том числе в работе с тестами и покрытием.





6.3. Выбор утилиты

В первую очередь необходимо ограничить набор поддерживаемых утилитой языков. Обратимся к исследованию компании TIOBE Inc за 2016 год. Согласно исследованию, первые пять мест занимают C-подобные языки, а также python. Ограничимся этими языками.

Jul 2016	Jul 2015	Change	Programming Language	Ratings	Change
1	1		Java	19.804%	+2.08%
2	2		C	12.238%	-3.91%
3	3		C++	6.311%	-2.33%
4	5	^	Python	4.166%	-0.09%
5	4	v	C#	3.920%	-1.73%
6	7	^	PHP	3.272%	+0.38%
7	9	^	JavaScript	2.643%	+0.45%
8	8		Visual Basic .NET	2.517%	+0.09%
9	11	^	Perl	2.428%	+0.62%
10	12	^	Assembly language	2.281%	+0.75%

Входные данные о покрытии и тестах должны иметь определенный стандартизированный формат, зависящий от анализатора покрытия,

используемого в проекте. Если в python данный анализатор, фактически, встроен в язык с помощью модуля coverage.py, то в остальных четырех случаях необходимо использование внешних утилит анализа. Нам необходимо выбрать наиболее популярную, удобную и совместимую с форматами, поддерживаемыми coverage.py.[10] Ниже рассмотрим подробнее конкретные решения.

6.4. Обзор решений для анализа покрытия кода

Так как в данном подисследовании делается упор на работу с компилируемыми языками, большинство решений будут представлять из себя пару компилятор-анализатор. Чтобы не заострять на этом внимание каждый раз, в общем случае данная связка работает так. Компилятор, который может являться расширением уже существующего для рабочей платформы, производит инструментацию и компиляцию кода. Анализатор же потребляет результат работы инъецированных инструкций. В следствии непосредственной интервенции в исполняемый код, скорость работы тестов падает, а размер файла увеличивается. Однако это необходимые и неизбежные затраты, когда приходится иметь дело с подобными языками.

6.4.1. Testwell CTC++

Мощная, но, с другой стороны, довольно простая утилита для анализа тестового покрытия кода на C и C++. С некоторыми расширениями, способна так же работать с C#. Представляет с собой анализатор, с помощью компиляции инструментированного файла представляющий отчет по выполнению тестов. Поддерживаемые виды оценки покрытия — вызовы функций, покрытие ветвей (с различными уровнями сложности), строковое покрытие и покрытие утверждений. Помимо того, что непосредственно интересует нас в нашей задаче, утилита предоставляет

дополнительные возможности, позволяющие оптимизировать сам тестируемый код, такие как поиск узких мест и долго исполняемого кода. Однако с точки зрения современного подхода к тестированию, Testwell CTC++ выглядит серьезно устаревшим. Как пример, отсутствует непосредственная связь между конкретным тестом и кодом, который был им охвачен.

6.4.2. Froglogic Coco

Более современный и мощный инструментарий, обладающий в несколько раз большим функционалом. Не смотря на то, что с точки зрения методов получения непосредственно информации о посещенных узлах данный набор утилит не сильно ушел вперед, он предоставляет серьезный потенциал для интеграции и различные форматы репортов. Но главное преимущество, особенно для нашего случая, это связка непосредственно тестов и покрытия. Для примера, анализ позволяет выявить оптимальный порядок запуска тестов для получения максимального покрытия, сравнение качества покрытия для разных патчей, а также, самое главное, возможность определить минимальное подмножество тестов, необходимых для проверки патча. К сожалению, новизна данного продукта, а также то, что в IT индустрии процесс миграции с заложенных в прошлом решений является достаточно трудным и, зачастую, признается не стоящим затраченных средств, не позволяет ему занять вполне заслуженное место лидера.[16]

6.4.3. Bullseye Covegare

“Корпоративный стандарт” для утилит тестирования, данный анализатор используется такими гигантами, как Hp, Philips и Abode. Продукт рассчитан на крупные проекты. Данная утилита позволяет работать с максимальным количеством различных вариантов оценки покрытия,

например, DataFlow Coverage, или специфической метрикой, как Race Coverage. И всего этого разработчики добились, работая с той же идеей инструктизации. Одним из преимуществ указывается простота использования, однако множественные настройки и скрипты делают его достаточно запутанным. Несмотря на это, возможность работы с отчетом о покрытии с помощью сторонних утилит, а также его распространенность, делает Bullseye Coverage перспективной целью для реализации поставленной оптимизационной задачи.[9]

6.4.4. GCT

Указываемая разработчиками Bullseye как единственная утилита, которая может соперничать с их продуктом в области реализации метрик, это свободное ПО было создано Брайеном Мариком (Brian Marick) в 1992 году и базируется на компиляторе gcc. Использовалась для тестирования ядра Unix. В данный момент морально устарела, не поддерживая большинство инноваций gcc вторых версий. Дальнейшая разработка в не ведется.

6.4.5. GCOV

Поставляется как стандартная утилита в составе пакета gcc и распространяется под лицензией GNU GPL. Многие проекты с открытым исходным кодом используют данное решение, что делает формат gcov перспективным для использования в реализации. Однако утилита предоставляет только самые базовые возможности оценки покрытия, например, покрытие по ветвям.[15]

6.4.6. Выбор утилиты

Мы остановились на двух возможных вариантах:

1. Bullseye Coverage
2. GCOV

Согласно обзору, Bullseye Coverage и GCOV являются наиболее распространенными утилитами, причем Bullseye является более коммерчески-ориентированным платным продуктом, а gcov - открытым и легкодоступным. С точки зрения экономического приложения, использование Bullseye является более выгодным, в том числе потому, что проекты, завязанные не него, являются крупными комплексными системами, для которых проблема загруженности тестами является довольно серьезной. С точки зрения разработчика утилиты, такие клиенты являются приоритетными, так как данного функционала в виде плагина не существует, а компании, уже купившие лицензию Bullseye, будут готовы купить и дополнение. С другой стороны, крупные компании при возникновении серьезной проблемы обычно стремятся решить ее как можно быстрее, чтобы минимизировать потерю времени и ресурсов. В таких случаях появляются внутренние решения для оптимизации процесса тестирования, что сужает применимость нашего оптимизационного решения.

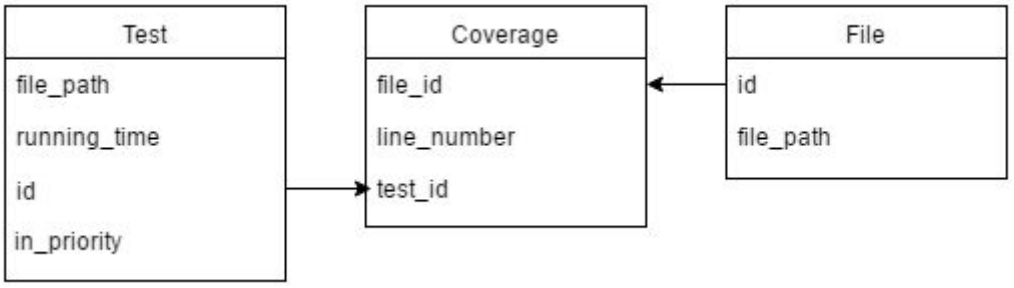
GCOV, в отличие от предыдущего решения, является бесплатной и, вследствие, более широко распространенной среди малых и средних проектов утилитой. Она куда проще и предоставляет меньше возможностей, чем Bullseye, но является признанной сообществом и надежной. Тренды к использованию более распространенных опенсорс решений в IT-компаниях, наоборот, расширяют область применения нашего решения относительно предполагаемой аудитории. Более того, формат результата работы GCOV совпадает с выдачей модуля coverage.py. Учитывая эти преимущества, для исследования будет использована именно эта утилита, однако поддержку Bullseye стоит рассматривать в

качестве возможного развития проекта в сторону монетизации и интеграции в существующие крупные продакшн-системы.

6.4.7. Интерфейс для выбранной утилиты

GCOV имеет два формата вывода - тестовый, а также XML-формат. Именно его мы и будем использовать. Основная задача - трансляция данных из XML в хранилище, о котором ниже. Используя наработки python-пакета lcovparse для работы с xml, создадим скрипт, который будет брать из репорта строки и помещать хэш текущего теста в хранилище.

6.5. Хранение данных о тестах и покрытии



Для начала, абстрактизируем данные. Каждому тесту необходимо сопоставить уникальный идентификатор, например, хеш из пути к нему (считаем, для простоты, что каждый тест размещен в отдельном файле). Основным инструментом для хранения данных о покрытии будет SQLite-хранилище с простой схемой, указанной выше. Имеем три таблицы: тесты, покрытие и файлы. В данной схеме хранится все, что требуется для работы алгоритма, а работа с данными - упрощается. Например, чтобы получить вес теста, посчитаем количество строк в таблице Coverage, имеющих данный test_id, и разделим на Test.running_time.

6.6. Сопоставление покрытия и изменений

Опишем все возможные интеракции между данными о покрытии и кодовой базой.

1. Удаление теста. Удаляем тест. В этом случае каскадом будут удалены данные о покрытии из таблицы Coverage.
2. Добавление теста. Добавляем запись в таблицу Test, а также данные о покрытии в таблицу Coverage после его запуска.
3. Изменение теста. Фактически, равносильно удалению старого теста и добавлению нового - покрытие могло значительно измениться. Удаляем данные о старом тесте, добавляем данные о новом.
4. Изменение кода. Согласно алгоритму, будут запущены тесты, которые покрывали этот код, но только те, что помещаются в указанное ограничение по времени. Данные о покрытии для этих тестов будут обновлены. Данные о покрытии для тестов, что запущены не были, останутся прежними. При удалении строк, данные о них удаляются. Однако требуется произвести сдвиг других строк на место удаленных.

6.7. Вычисление разницы между областями покрытия

Как было указано ранее, в качестве хранилища мы будем использовать sqlite базу с одной таблицей. Так как алгоритм вычисления уникального покрытия является последовательным, то все уникальные строки для теста k можно получить с помощью выбора всех строк из таблицы Coverage, для которых test_id будет равен k, если не существует строки с тем же номером, для которых test_id принадлежит множеству рассмотренных ранее тестов.

6.8. Интерфейс для работы с тестирующей системой

Для запуска тестов на языке C требуется перекомпилировать программу. GNU make является одной из наиболее популярных утилит для автоматизации сборки и тестирования проектов.[12] Внедрение утилиты должно ложиться на плечи сотрудника, в должностные обязанности

которого входит разработка и внедрение систем автоматизации и непрерывной интеграции. Альтерация Makefile - действие достаточно простое, однако из-за отсутствия общего стандарта по поиску и запуску тестов, предоставить универсальное решение по интеграции является достаточно сложной задачей. Для упрощения данного процесса, утилита должна быть понятной для разработчика любого уровня и зависеть только от своей базы данных. Это означает, что для ее внедрения требуется минимальное количество усилий:

1. Инициализация
2. Изменение Makefile с полного запуска тестов на запуск только тех тестов, что перечислены в результате работы утилиты.
3. Периодическое обновление полной информации о покрытии

Благодаря простоте схемы базы данных, весь процесс обновления утилиты заключается в замене исполняемого файла. В случае, если формат БД будет изменен, существуют доступные решения по безболезненной миграции данных, например, alembic.

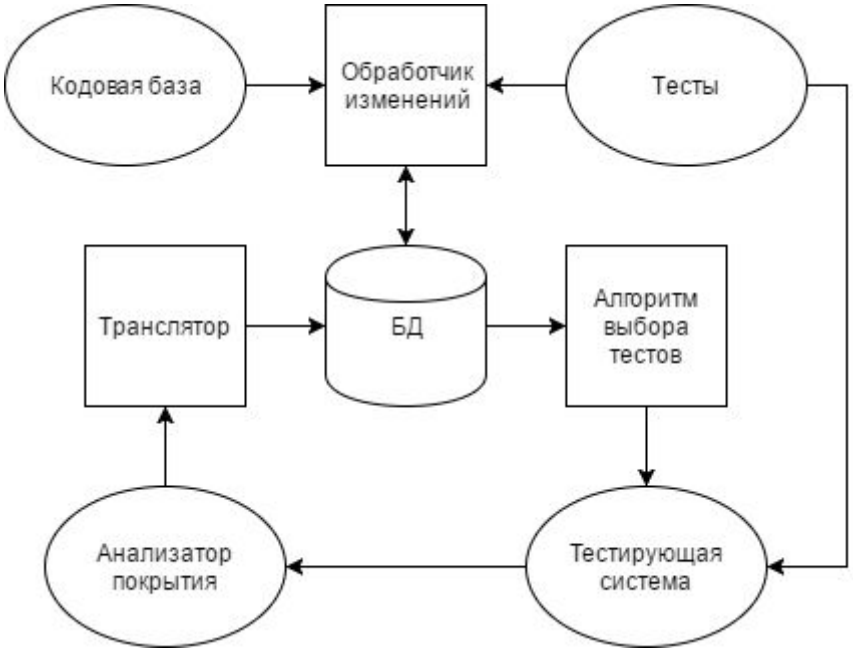
6.9. Инициализация и поддержание актуальности своих данных

Инициализация производится путем получения транслятором полной информации о покрытии, то есть последовательности пар путей к тестам и соответствующих им результатов работы анализатора.

После каждого запуска утилиты необходимо обновить информацию о покрытии запущенных тестов. Так как приоритет при запуске получают тесты, покрывающие наибольшее количество строк за наименьшее время, то средняя консистентность данных будет меняться достаточно медленно. Как мы указывали ранее, такие “оптимальные” тесты обычно являются общими проверками, быстро проверяющими основной функционал. Вероятность того, что объем затронутого кода для тестов будет уменьшен

- достаточно низка, из чего следует, что большая часть записей о покрытии будет оставаться актуальной. Однако для точности и полной проверки работоспособности продукта требуется периодически запускать все тесты. Одной из функций утилиты может быть предоставление оптимального периода полного тестирования, исходя из скорости устаревания базы.

6.10. Полная схема утилиты



Постараемся достичь максимальной модульности и обособленности составных частей утилиты. Это также упростит тестирование ее самой.

6.10.1. Транслятор

Принимает в качестве входных параметров результат работы анализатора покрытия (в конкретном случае - xml выдачу gcov). Из входных данных извлекаются данные о строках, а также путь к тесту. Данные о тесте удаляются, после чего записываются новые.

6.10.2. Обработчик изменений

На вход получает выдачу в формате утилиты GNU diff (например, результат работы git diff).[17] Помечает все тесты как обычные. В случае удаления строк удаляет строки в базе. Сдвигает строки в случае

добавления, но не добавляет новых записей. Описанным ранее способом выделяет множество тестов для покрытия изменений. Помечает выбранные тесты как приоритетные.

6.10.3. Алгоритм выбора тестов

Данный модуль работает только с базой и заданным ограничением по времени. Результатом является список путей к рекомендуемым для исполнения тестам. Алгоритм работает в два этапа - сначала выбираются попадающие в интервал тесты из помеченных, после чего происходит добор из оставшихся.

6.11. Рекомендации по использованию и ограничения

1. Независимость тестов с точки зрения логики и порядка запуска. Хотя повторение именно логической части теста не помешает работе утилиты, ее эффективность будет ниже. Зависимость же тестов от порядка запуска полностью отметает возможность выбора конкретных тестов с соблюдением их валидности.
2. Отсутствие метапрограммирования, а также зависимости результата работы утверждения от его позиции в коде. В этих случаях оценка последствий изменений, опирающаяся лишь на данные о покрытии и позициях строк, перестает работать в принципе.
3. Утилиту рекомендуется использовать на уровне конкретных разработчиков, желающих удостовериться, что их изменения не повлекли за собой обширных проблем. Это знание является серьезным фактором для процесса ревью, для уверенности разработчика в своих действиях, а также спасает от бесполезной траты машинного времени.
4. Периодический запуск всех тестов. Для обеспечения качества продукта этого не избежать. Предлагается работа в двух циклах

тестирования. Малый цикл - per-commit запуск оптимального набора тестов для снижения шанса попадания ошибок в билд. Большой цикл - per-build запуск всего тестового набора для доказательства валидности накопленных изменений.

5. Многие VCS позволяют делать такую вещь, как коммит-хук - запуск скриптов при коммите. Интегрировав утилиту подобным образом в сервер системы, можно снизить количество невалидных пулл-реквестов, позволяя сократить время, потраченное разработчиками на поиск потенциальных ошибок.
6. Требования по работе с окружением. Для каждого окружения необходимо поддерживать собственную базу данных, так как в зависимости, например, от флагов компиляции, данные о покрытии будут разниться.

7. Экономическая эффективность

При рассмотрении ситуации, решением которой является полученная оптимизация, будет получена выгода в виде сэкономленных часов работы над проектом. Обычно, если наличествует длительный этап, не требующий непосредственного участия человека, менеджмент назначает работнику несколько задач одновременно. Но даже при работе над несколькими правками кода в параллели, в какой-то момент разработчик будет все-равно заблокирован тестами, вследствие чего не сможет продолжать эффективную работу.

Рассмотрим два исходных случая.

1. Процесс непрерывной интеграции не налажен. В таком случае тестирование проводится по требованию, обычно перед сборкой. Велики риски нахождения проблем только на конечном этапе цикла разработки. Утилита позволяет внедрить практику непрерывной интеграции, существенно снижая риски, а также потенциальные траты ресурсов на экстренные исправления.
2. Процесс непрерывной интеграции налажен. Разработчик вынужден ждать непропорционально огромное время, чтобы продолжить работу. Оптимизационная утилита позволяет сместить длительный автоматический этап на нерабочее время (например, ночь, или выходные), при этом сохраняя выгоды от непрерывного тестирования. В данном случае повышается эффективность работы

программистов, а также снижаются затраты на непосредственно тестирование.

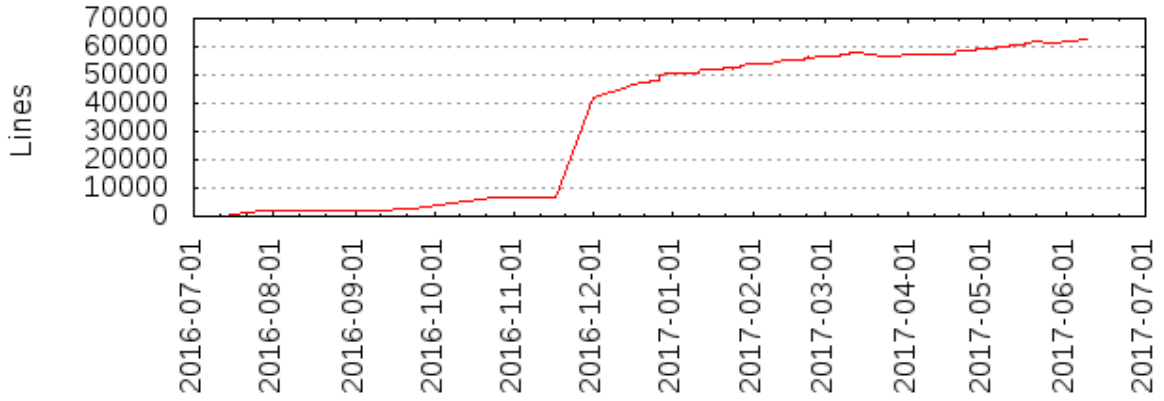
8. Прикладные расчеты и результаты

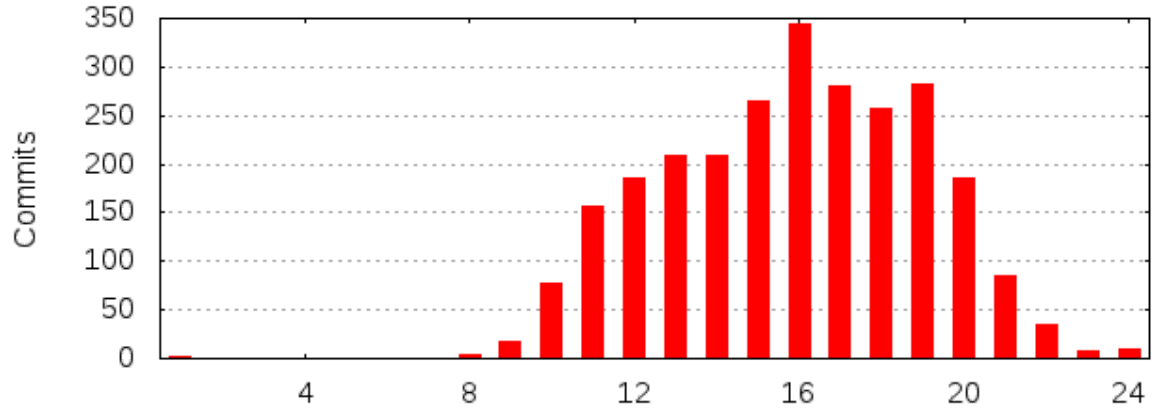
Проверим на существующем проблемном проекте исследуемый алгоритм оптимизации.

8.1 Проект

В качестве основы для проведения расчетов эффективности был взят репозиторий одного из компонент продукта Acronis Cloud Platform. В данном репозитории используется язык python, для которого интеграция с утилитой не требует дополнительных решений, что послужило основной причиной его выбора. Данный продукт является проприетарным, так что в исследовании будут продемонстрированы только статистические данные. Для их получения используются базовые возможности git. В статистике указывается лишь фактическое состояние ветки master на данный момент в силу удаления ветвей после успешного слияния, то есть мы располагаем данными только о коммитах, прошедших тестирование и слитых в master. Ниже представлена необходимая информация о проекте, а также графики роста репозитория и распределение коммитов по времени суток.

Статистика репозитория		
Объем кода	**Объем тестов**	**Тестовый цикл**
13740 строк	18810 строк	~120 минут
Коммиты	**Статистика изменений**	**Статистика коммитов**
2678 (1834)	55 строк/коммит	~9 в рабочий день
Изменения кода	**Изменения тестов**	**Время разработки**
64220 строк	36987 строк	282 рабочих дня





8.2 Расчеты

Рассчитаем, сколько времени было потрачено на тестирование.

Проект использует jenkins в качестве системы непрерывной интеграции. Запуск тестов происходит для каждого коммита в pull-request.[18] Согласно статистике ci-системы, в 36% случаев тесты были провалены. Итого получим примерно 1915 запусков. Заметим, что тестирование происходит в рабочее время, а также то, что каждый третий коммит посвящен исправлению ошибок предыдущего коммита. Команда располагает 4 серверами для параллельного запуска тестов. Учтем, что работа производится по системе TDD, описанной ранее, что уменьшает вероятность внесения изменений в ранее протестированные и слитые части проекта.

Как указывалось ранее, мы не можем просто аппроксимировать времена работы ci-задач, так как система изменялась. Поэтому рассчитаем функциональную зависимость времени работы тестов от объема кода. Для языка python хорошей практикой для написания тестов является использование setUp и tearDown методов, которые ответственны за создание и уничтожение тестового окружения. Для каждого теста запускается соответствующая ему пара. Однако, несмотря на это, статистически время работы тестов растет линейно - а именно по 4 секунды на коммит. Подсчитаем общее потраченное на тестирование время:

$$\frac{1915}{844}\int_{0}^{844} 4\,x = 3232520 \text{ секунд} \approx 897 \text{ часов}$$

Итого, имеем следующие данные для оценки исследуемого алгоритма оптимизации:

Вероятность	Общее время	Часов в день на	Эффективные

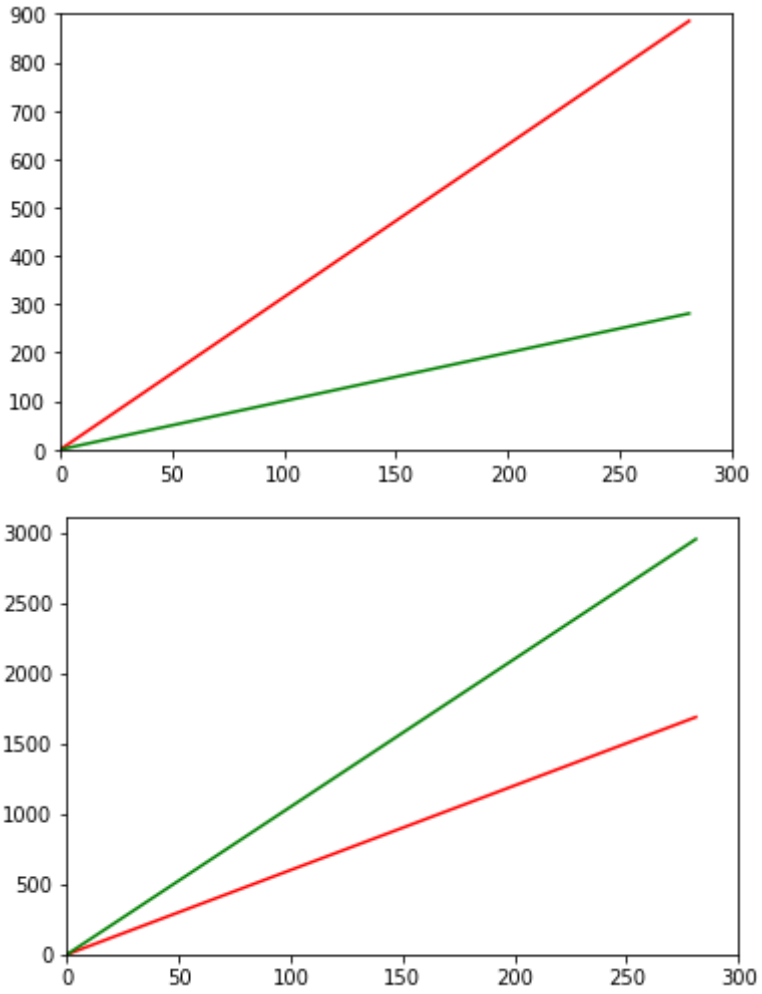
ошибки	тестирования	тестирование	коммиты
36%	897 часов	3.15	6 из 9

8.3 Применение оптимизации

Теперь рассмотрим тот же проект, но с учетом оптимизации. Ограничим время работы тестов 20 минутами. Подобное ограничение достигается за счет большого количества тестов, покрывающих от 45% кода. Более того, в ходе исследования выяснилось, что количество тестов, покрывающих менее 40% кода, пренебрежимо мало. Пока общее время работы тестов не выходит за рамки ограничения, очевидно, никаких отличий не будет. Далее будем запускать только оптимальные тесты в рабочее время, а накопившиеся коммиты - ночью. Запуская меньше тестов, мы повышаем риск упустить ошибку, однако повышаем количество “целевых” изменений кода, не связанных с исправлением ошибок. Освободившееся время позволяет команде внести 12 коммитов. Вероятность провала ночного теста близка к 100%. Однако в этом случае все ошибки могут быть исправлены одним коммитом.

8.4 Результаты

Вероятность ошибки	Общее время тестирования	Часов в день на тестирование	Эффективные коммиты
52%	282 часа	1 час	10-11 из 12



На рисунках выше показаны последовательно затраты на тестирование и количество эффективных коммитов в зависимости от времени при одинаковых условиях разработки. Красным цветом обозначена зависимость для неоптимального, а зеленым - для оптимального случая.

Как видно из исследования, эффективность работы команды выросла почти в два раза, а нагрузка на тестовые сервера упала в 4. Однако данный

случай не является критическим - ci все еще работает, а тесты проходят за адекватное время. В случае, если проблемы с длительностью тестового цикла куда более значимы, выгода от использования возрастает. Однако, если цикл тестирования занимает больше одной ночи, необходимо изменение рабочего процесса команды и синхронизация его с циклом полного тестирования. В такой ситуации эффект от восстановления процесса непрерывной интеграции будет первостепенно важным.

9. Вывод

Результатом этого исследования является алгоритм, а также наработки по его непосредственной реализации, способный серьезно упростить и ускорить процесс разработки программного обеспечения, а также восстановить процесс непрерывной интеграции для объемных проектов. На примере существующего репозитория, для которого рассмотренные проблемы были актуальны, показаны преимущества использования разработанного метода тестирования по сравнению с классической схемой. Произведена оценка эффективности и связанных с применением метода рисков. Были выявлены недостатки первоначальной идеи, определены ограничения применения выработанного решения. Однако, наиболее важным результатом работы можно считать то, что в очередной раз было подтверждено, что идеального со всех сторон решения проблемы не существует. Невозможно, в конечном итоге, получить качественный продукт, если в ходе разработки приоритизировать решение одних важных проблем над другими.

10. Список литературы

1. Ron Patton. Software Testing
2. Гленфорд Майерс, Том Баджетт, Кори Сандлер. Искусство тестирования программ, 3-е издание
3. Бейзер Б. Тестирование чёрного ящика. Технологии функционального тестирования программного обеспечения и систем
4. McConnell, Steve. Code Complete (2nd ed.)
5. Dustin, Elfriede. Effective Software Testing
6. Kolawa, Adam; Huizinga, Dorota. Automated Defect Prevention: Best Practices in Software Management
7. Kaner, Cem. "Exploratory Testing"
8. Bach, James. "Risk and Requirements-Based Testing"
9. Bullseye Testing Technology. "Intermediate Coverage Goals"
10. Python Documentation
11. Xie, Tao. "Towards a Framework for Differential Unit Testing of Object-Oriented Programs"
12. Robert Mecklenburg. Managing projects with gnu make third edition
13. Hayhurst, Kelly; Veerhusen, Dan; Chilenski, John; Rierson, Leanna. "A Practical Tutorial on Modified Condition/ Decision Coverage"
14. Fowler, Martin. "Continuous Integration"
15. Using the GNU Compiler Collection
16. https://www.froglogic.com/coco/
17. https://www.gnu.org/software/diffutils/
18. https://jenkins.io/
19. http://git-scm.com
20. ГОСТ Р ИСО/МЭК 25010-2015
21. https://www.tiobe.com/tiobe-index/

22. Kellerer, Hans, Pferschy, Ulrich, Pisinger, David; Knapsack Problems