**КОНТРОЛЬНАЯ САМОСТОЯТЕЛЬНАЯ РАБОТА №1.**

Вариант 15. Теоретическая часть

«Регрессионное тестирование: методики, не связанные с отбором тестов и методики порождения тестов»

**Выполнил:**

студент 2 курса 14 группы

факультета прикладной математики и информатики

*Петров Андрей Александрович*

Изменения в коде неизбежно сопровождают процесс разработки программного обеспечения. Добавляется новая функциональность, вносятся изменения в существующую, устраняются дефекты. При этом любые изменения могут затронуть уже существующую функциональность, которая исправно работала. Проверить, чтобы изменения не «поломали» рабочее ПО, – задача регрессионного тестирования.

**Цель регрессионного тестирования** – удостовериться в том, что существующая функциональность не была затронута изменениями в коде.

Например, команда разработала простое приложение-фонарик. В приложении есть всего две функции: включение и выключение фонарика. Лицо проводит тестирование функциональности, чтобы убедиться в правильной работе этих функций и приложения в целом. Результат тестирования положительный – не о чем беспокоиться.

Спустя какое-то время команда решает усовершенствовать приложение, добавив в него функцию включения от того, что пользователь встряхивает телефон. Новая функциональность добавлена, проводится очередной раунд тестирования. Однако на этот раз необходимо проверить работоспособность не только новой функциональности, но и включения/выключения. А вдруг изменения затронули и её каким-то образом?

Этот пример демонстрирует место регрессионного тестирования в процессе разработки ПО.

**Интеграционное регрессионное тестирование**

С появлением новых направлений в разработке программного обеспечения (например, объектно-ориентированного программирования), поощряющих использование большого количества маленьких процедур, повышается важность обработки межмодульного влияния изменений кода для методик уменьшения стоимости регрессионного тестирования. Для решения этой задачи необходимо рассматривать зависимости по глобальным переменным, когда переменной в одной или нескольких процедурах присваивается значение, которое затем используется во многих других процедурах. Такую зависимость можно рассматривать как зависимость между процедурами по потоку данных. Также возможны межмодульные зависимости по ресурсам, например по памяти, когда ресурс разделяется между несколькими процедурами. Отметим, что при системном регрессионном тестировании зависимости такого рода можно игнорировать.

Если изменение спецификации требований затрагивает глобальную переменную, могут потребоваться новые модульные тесты. В противном случае, повторному выполнению подлежат только модульные тесты, затрагивающие как измененный код, так и операторы, содержащие ссылку на глобальную переменную.

Брандмауэр можно определить как подмножество графа вызовов, содержащее измененные и зависящие от изменений процедуры и интерфейсы. Методы отбора тестов, использующие брандмауэр, требуют повторного интеграционного тестирования только тех процедур и интерфейсов, которые непосредственно вызывают или вызываются из измененных процедур.

**Регрессионное тестирование объектно-ориентированных программ**

Объектно-ориентированный подход стимулирует новые приложения методик выборочного повторного тестирования. Действительно, при изменении класса необходимо обнаружить в наборе тестов класса только тесты, требующие повторного выполнения. Точно так же при порождении нового класса из существующего необходимо определить тесты из множества тестов базового класса, требующие повторного выполнения на классе-потомке. Хотя благодаря инкапсуляции вероятность ошибочного взаимодействия объектно-ориентированных модулей кода уменьшается, тем не менее, возможно, что тестирование прикладных программ выявит ошибки в методах, не найденные при модульном тестировании методов. В этом случае необходимо рассмотреть все прикладные программы, использующие измененный класс, чтобы продемонстрировать, что все существующие тесты, способные обнаруживать ошибки в измененных классах, были запущены повторно и было выбрано безопасное множество тестов. При повторном тестировании прикладных программ, классов или их наследников применение методик выборочного повторного тестирования к существующим наборам тестов может принести немалую пользу.

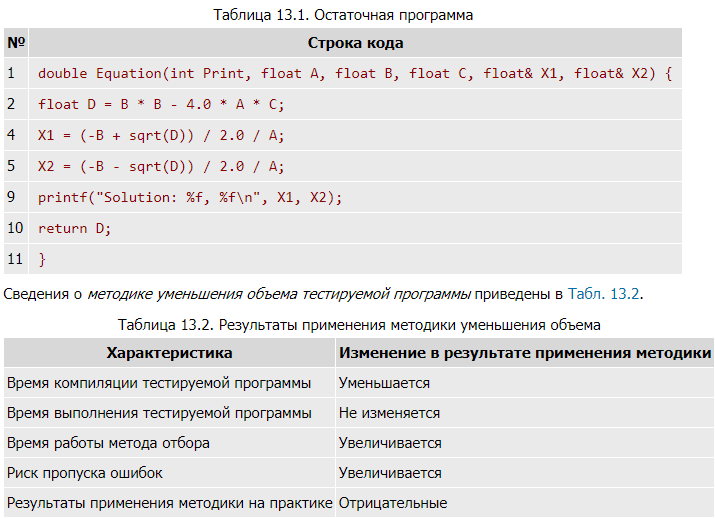
В объектно-ориентированной программе вызов метода во время выполнения может быть сопоставлен любому из ряда методов. Для заданного вызова мы не всегда можем статически определить метод, с которым он будет связан. Выборочные методы повторного тестирования, которые полагаются на статический анализ, должны обеспечивать механизмы для разрешения этой неопределенности.

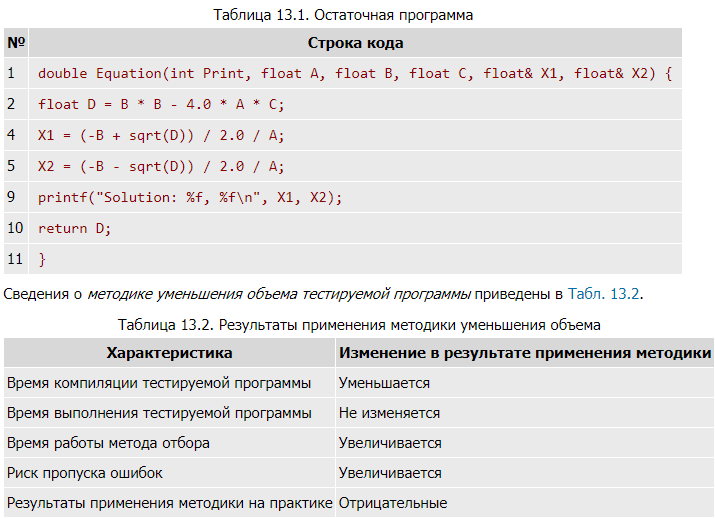
**Уменьшение объема тестируемой программы**

Еще один путь сокращения затрат на регрессионное тестирование состоит в том, чтобы вместо повторного тестирования (большой) измененной программы с использованием соответственно большого числа тестов доказать, что измененная программа адекватно тестируется с помощью выполнения некоторого (меньшего) числа тестов на остаточной программе. Остаточная программа создается путем использования графа зависимости системы вместо графа потока управления, что позволяет исключить ненужные зависимости между компонентами в пределах одного пути графа потока управления. Так, корректировка какого-либо оператора в идеале должна приводить к необходимости тестировать остаточную программу, состоящую из всех операторов исходной программы, способных повлиять на этот оператор или оказаться в сфере его влияния. Для получения остаточной программы необходимо знать место корректировки (в терминах операторов), а также информационные и управляющие связи в программе. Этот подход работает лучше всего для малых и средних изменений больших программ, где высокая стоимость регрессионного тестирования может заставить вообще отказаться от его проведения. Наличие дешевого метода остаточных программ, обеспечивающего такую же степень покрытия кода, делает регрессионное тестирование успешным даже в таких случаях.

Метод остаточных программ имеет ряд ограничений. В частности, он не работает при переносе программы на машину с другим процессором или объемом памяти. Более того, он может давать неверные результаты и на той же самой машине, если поведение программы зависит от адреса ее начальной загрузки, или если для остаточной программы требуется меньше памяти, чем для измененной, и, соответственно, на остаточной программе проходит тест, который для измененной программы вызвал бы ошибку нехватки памяти. Исследования метода на программах небольшого объема показали, что выполнение меньшего количества тестов на остаточной программе не оправдывает затрат на отбор тестов и уменьшение объема программы. Однако для программ с большими наборами тестов это не так.

Для теста 1 рис. 12.1 для функции Equation остаточная программа выглядит так, как показано в Табл. 13.1. Нумерация строк оставлена такой же, как в исходной программе. Таким образом, можно заметить, что были удалены строки 6 и 7, которые не затрагиваются тестом 1 в ходе его выполнения, а также строки 3 и 8, содержащие вычисление предикатов, которые в ходе выполнения теста всегда истинны. Запуск теста на полной измененной программе и на остаточной программе приводит к активизации одних и тех же операторов, поэтому выигрыша во времени получить не удается, однако за счет сокращения объема программы уменьшается время компиляции. Для нашего примера этот выигрыш незначителен и не оправдывает затрат на анализ, необходимый для уменьшения объема. Таким образом, рассмотренная технология рекомендуется к применению, прежде всего, в случаях, когда стоимость компиляции относительно высока.





**Методы упорядочения**

Методы упорядочения позволяют инженерам-тестировщикам распределить тесты так, что тесты с более высоким приоритетом исполняются раньше, чем тесты с более низким приоритетом, чтобы затем ограничиться выбором первых n тестов для повторного исполнения. Это особенно важно для случаев, когда тестировщики могут позволить себе повторное выполнение только небольшого количества регрессионных тестов.

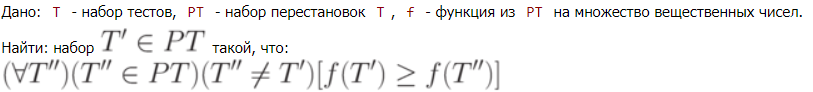
Одной из проблем упорядочения тестов является отсутствие представлений о том, скольких тестов в конкретном проекте достаточно для отбора. В отличие от подхода минимизации, использующего все тесты минимального набора, оптимальное число тестов в упорядоченном наборе неизвестно. Возникает проблема баланса между тем, что необходимо делать в ходе регрессионного тестирования, и тем, что мы можем себе позволить. В итоге количество запускаемых тестов определяется ограничениями по времени и бюджету и порядком тестов в наборе. С учетом этих факторов следует запускать повторно как можно больше тестов, начиная с верхней строки списка упорядоченных тестов.

Возможное преимущество упорядочения тестов состоит в том, что сложность соответствующего алгоритма, O(n2) в наихудшем случае, меньше, чем сложность алгоритма минимизации, который в некоторых случаях может требовать экспоненциального времени выполнения.

Проблему упорядочения тестов можно сформулировать следующим образом:

Дано: T - набор тестов, PT - набор перестановок T, f - функция из PT на множество вещественных чисел.

Найти: набор Т’∈ РТ такой, что:

В приведенном определении PT представляет собой множество всех возможных вариантов упорядочения - T, а f – функция, которая, будучи применена к любому такому упорядочению, выдает его вес. (Предположим, что большие значения весов предпочтительнее малых.)

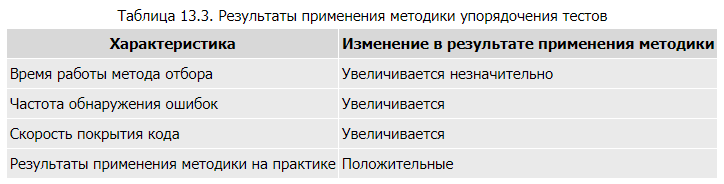
Упорядочение может преследовать различные цели:

* Увеличение частоты обнаружения ошибок наборами тестов, то есть увеличение вероятности обнаружить ошибку раньше при выполнении регрессионных тестов из этих наборов.
* Ускорение процесса покрытия кода тестируемой системы и достижение требуемой степени покрытия кода на более ранних этапах процесса тестирования.
* Быстрейший рост вероятности того, что тестируемая система надежна.
* Увеличение вероятности обнаружения ошибок, связанных с конкретными изменениями кода, на ранних этапах процесса тестирования и т.п.

Методы упорядочения планируют выполнение тестов в процессе регрессионного тестирования в порядке, увеличивающем их эффективность в терминах достижения заданной меры производительности. Обоснование использования упорядочивающего метода состоит в том, что упорядоченный набор тестов имеет большую вероятность достижения цели, чем тесты, расположенные по какому-либо другому правилу или в случайном порядке.

Различают два типа упорядочения тестов: общее и в зависимости от версии. Общее упорядочение тестов при данных программе P и наборе тестов T подразумевает нахождение порядка тестов T, который окажется полезным для тестирования нескольких последовательных измененных версий. Считается, что для этих версий итоговый упорядоченный набор тестов позволяет в среднем быстрее достигнуть цели, ради которой производилось упорядочение, чем исходный набор тестов. Однако имеется значительный объем статистических свидетельств в пользу наличия связи между частотой обнаружения ошибок и конкретной версией тестируемой программы: разные версии программы предоставляют различные возможности по упорядочению тестов. При регрессионном тестировании мы имеем дело с конкретной версией программного продукта и хотим упорядочивать тесты наиболее эффективно по отношению именно к этой версии.

Например, упорядочивать тесты можно по количеству покрываемых ими изменений кода. При безопасном отборе тестов из рис. 12.1 будут выбраны тесты 1, 2 и 5, из которых наиболее приоритетным является тест 2, так как он затрагивает оба изменения, тогда как тесты 1 и 5 – только одно.



**Методики порождения новых тестов**

Порождение новых тестов при структурном регрессионном тестировании обычно обусловлено недостаточным уровнем покрытия. Новые тесты разрабатываются так, чтобы задействовать еще не покрытые участки исходного кода. Процесс прекращается, когда уровень покрытия кода достигает требуемой величины (например, 80%). Разработка новых тестов при функциональном регрессионном тестировании является менее тривиальной задачей и обычно связана с вводом новых требований либо с желанием проверить некоторые сценарии работы системы дополнительно.

Основой большинства программных продуктов для управляющих применений, находящихся в промышленном использовании, является цикл обработки событий. Сценарий работы с системой, построенной по такой архитектуре, состоит из последовательности транзакций, управление после обработки каждой транзакции вновь передается циклу обработки событий. Выполнение транзакции приводит к изменению состояния программы; в результате некоторых транзакций происходит выход из цикла и завершение работы программы. Тесты для таких программ представляют собой последовательность транзакций.

Развитие программного продукта от версии к версии влечет за собой появление новых состояний. Поскольку большинство тестов легко может быть расширено путем добавления дополнительных транзакций в список, новые тесты можно создавать путем суперпозиции уже имеющихся, с учетом информации об изменении состояния тестируемой системы в результате прогона теста. Этот подход позволяет указать, какого рода новые тесты с наибольшей вероятностью обнаружат ошибки.

Обозначим тестируемую программу P, а множество ее тестов T = {t1, t2, …, tn}. Будем считать, что состояние тестируемой программы s определяется совокупностью значений некоторого подмножества глобальных и локальных переменных. При создании новых тестов будем рассматривать состояния программы перед запуском теста (s0) и после его окончания (sj). Информацию об этих состояниях необходимо собирать для каждого теста по результатам запуска на предыдущей версии продукта.

Методика порождения новых тестов на основе анализа "подозрительных" состояний сводится к описанной ниже последовательности действий.

1. Вычисление списка глобальных и локальных переменных, определяющих состояние программы s.
2. Сбор информации (на основе анализа профиля программы, полученного на предыдущей версии продукта i-1, для каждого существующего теста tj) о состояниях программы перед запуском теста и после его окончания (т.е. s0 и sj). Множество таких состояний обозначается 
3. Выполнение на текущей версии продукта i новых и выбранных регрессионных тестов из множества T’. По аналогии с Si–1 вычисляется множество Si, которое сохраняется под управлением системы контроля версий.
4. Оценка "подозрительных" с точки зрения наличия ошибок множества новых по сравнению с предыдущими версиями состояний Ni в соответствии со следующей формулой: Ni = Si \ Si - 1
5. Анализ состояний множества Ni, в которых дальнейшая работа продукта невозможна в соответствии со спецификацией. Предмет анализа - определить создаются ли эти состояния в результате выполнения тестов, проверяющих нештатные режимы работы продукта, или каких-либо других тестов. В последнем случае фиксируется ошибка.
6. Исключение нештатных состояний из множества Ni.
7. Переход к шагу 10, если новых состояний, допускающих продолжение выполнения программы, не обнаружено, т.е. 
8. Для каждого состояния множества Ni вычисление вектора отличия от исходного состояния s0, т.е. множества переменных, измененных по сравнению с s0.
9. Модификация множества измененных строк исходного кода P на основе информации об измененных переменных и использование какой-либо методики отбора тестов для выборочного регрессионного тестирования.
10. Повторное выполнение шагов 3-9 до достижения состояния либо до истечения времени, отведенного на регрессионное тестирование. Использование методов разбиения на классы эквивалентности для досрочного принятия решения о прекращении цикла тестирования, если ни один из тестов, созданных на очередном этапе, не принадлежит к новому классу эквивалентности.

Для приведенной методики организации тестирования, когда новые тесты получаются в результате суперпозиции уже имеющихся, целесообразно в качестве исходных тестов, т.е. тех "кирпичиков", из которых будут строиться тесты в дальнейшем, брать тесты, заключающие всего одну элементарную проверку. Это помогает избежать избыточности при многократном слиянии тестов, когда искомые "подозрительные" ситуации возникают в ходе работы теста, но не анализируются, так как не повторяются при его завершении.

Использование описанной методики позволяет в программном комплексе находить ошибки, не обнаруживаемые исходным набором тестов. Отметим, что применение предложенного подхода невозможно для программ, понятие состояния для которых не определено.