**МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ**

**БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ**

**Факультет прикладной математики и информатики**

**Кафедра информационных систем управления**

**Реферат по теме «Тестирование путем покрытия логики**

**программы (“белый ящик”, “серый ящик”)»**

Выполнил студент *1* курса *13* группы

*Гришкин Андрей Иванович*

г. Минск

2019

**Оглавление**

[**Введение** 1](#_Toc979445)

[**Проектирование тестов** 1](#_Toc979446)

[**Методологии “белый ящик”** 2](#_Toc979447)

[**Покрытие операторов** 3](#_Toc979448)

[**Покрытие решений** 4](#_Toc979449)

[**Покрытие условий** 5](#_Toc979450)

[**Покрытие решений/условий** 6](#_Toc979451)

[**Комбинаторное покрытие условий** 7](#_Toc979452)

[**Преимущества и недостатки** 9](#_Toc979453)

[**Методологии “серый ящик”** 10](#_Toc979454)

[**Регрессионное тестирование** 10](#_Toc979455)

[**Шаблонное тестирование** 10](#_Toc979456)

[**Преимущества и недостатки** 11](#_Toc979457)

[**Заключение** 11](#_Toc979458)

[**Список литературы** 12](#_Toc979459)

# **Введение**

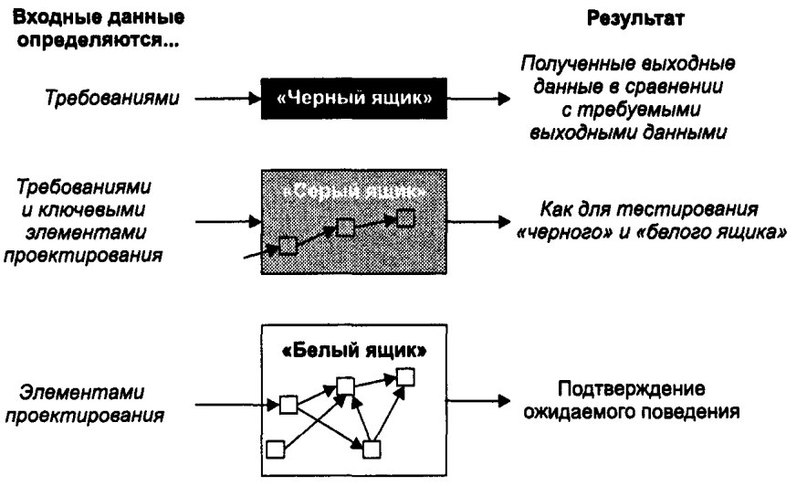
В данной работе будут рассмотрены классические методы, которые относятся к тестированию путем покрытия логики программы. Это методы, которые предназначены для тестирования не программного комплекса в целом, а для тестирования, прежде всего, программного кода. Понимание данных методов позволяет оценивать остальные методы с точки зрения полноты тестирования и подхода к тестированию.

# **Проектирование тестов**

Тест для любой программы будет обязательно неполным (иными словами, тестирование не может гарантировать отсутствия всех ошибок). Поэтому главной целью любой стратегии проектирования является уменьшение этой «неполноты» тестирования настолько, насколько это возможно.

Если ввести ограничения на время, стоимость, машинное время и т. п., то ключевым вопросом тестирования становится следующий: «Какое подмножество всех возможных тестов имеет наивысшую вероятность обнаружения большинства ошибок?» (Степанченко, 2006)

Изучение методологий проектирования тестов дает ответ на этот вопрос. По-видимому, наихудшей из всех методологий является тестирование со случайными входными значениями (стохастическое) – процесс тестирования программы путем случайного выбора некоторого подмножества из всех возможных входных величин. В терминах вероятности обнаружения большинства ошибок случайно выбранный набор тестов имеет малую вероятность быть оптимальным или близким к оптимальному подмножеству. В данной работе рассматриваются несколько подходов, которые позволяют более разумно выбирать тестовые данные. Исчерпывающее тестирование по принципу черного или белого ящика в общем случае невозможно. Однако при этом отметим, что приемлемая стратегия тестирования может обладать элементами обоих подходов. Таковой является стратегия, излагаемая в этой работе. Можно разработать довольно полный тест, используя определенную методологию проектирования, основанную на принципе черного ящика, а затем дополнить его проверкой логики программы (т. е. с привлечением методов стратегии белого ящика).



# **Методологии “белый ящик”**

Все методологии, обсуждаемые в настоящей работе можно разделить

− покрытие операторов;

− покрытие решений;

− покрытие условий;

− покрытие решений/условий. (Майерс, 1982)

Хотя перечисленные методы будут рассматриваться здесь по отдельности, при проектировании эффективного теста программы рекомендуется использовать если не все эти методы, то, по крайней мере, большинство из них, так как каждый метод имеет определенные достоинства и недостатки (например, возможность обнаруживать и пропускать различные типы ошибок). Правда, эти методы весьма трудоемки, поэтому некоторые более опытные лица, ознакомившись с ними, могут не согласиться с данной рекомендацией. Однако следует представлять себе, что тестирование программы – чрезвычайно сложная задача. Для иллюстрации этого приведу известное изречение: «Если вы думаете, что разработка и кодирование программы – вещь трудная, то вы еще ничего не видели» (Майерс, 1980.). Рекомендуемая процедура заключается в том, чтобы разрабатывать тесты, используя стратегию черного ящика, а затем как необходимое условие – дополнительные тесты, используя методы белого ящика.

Тестирование по принципу белого ящика характеризуется степенью, в какой тесты выполняют или покрывают логику (исходный текст) программы.

## **Покрытие операторов**

Если отказаться полностью от тестирования всех путей, то можно показать, что критерием покрытия является выполнение каждого оператора программы, по крайней мере, один раз. Это метод покрытия операторов. К сожалению, это слабый критерий, так как выполнение каждого оператора, по крайней мере, один раз есть необходимое, но недостаточное условие для приемлемого тестирования по принципу белого ящика (рис. 3).

Предположим, что на рис. 3 представлена небольшая программа, которая

должна быть протестирована.

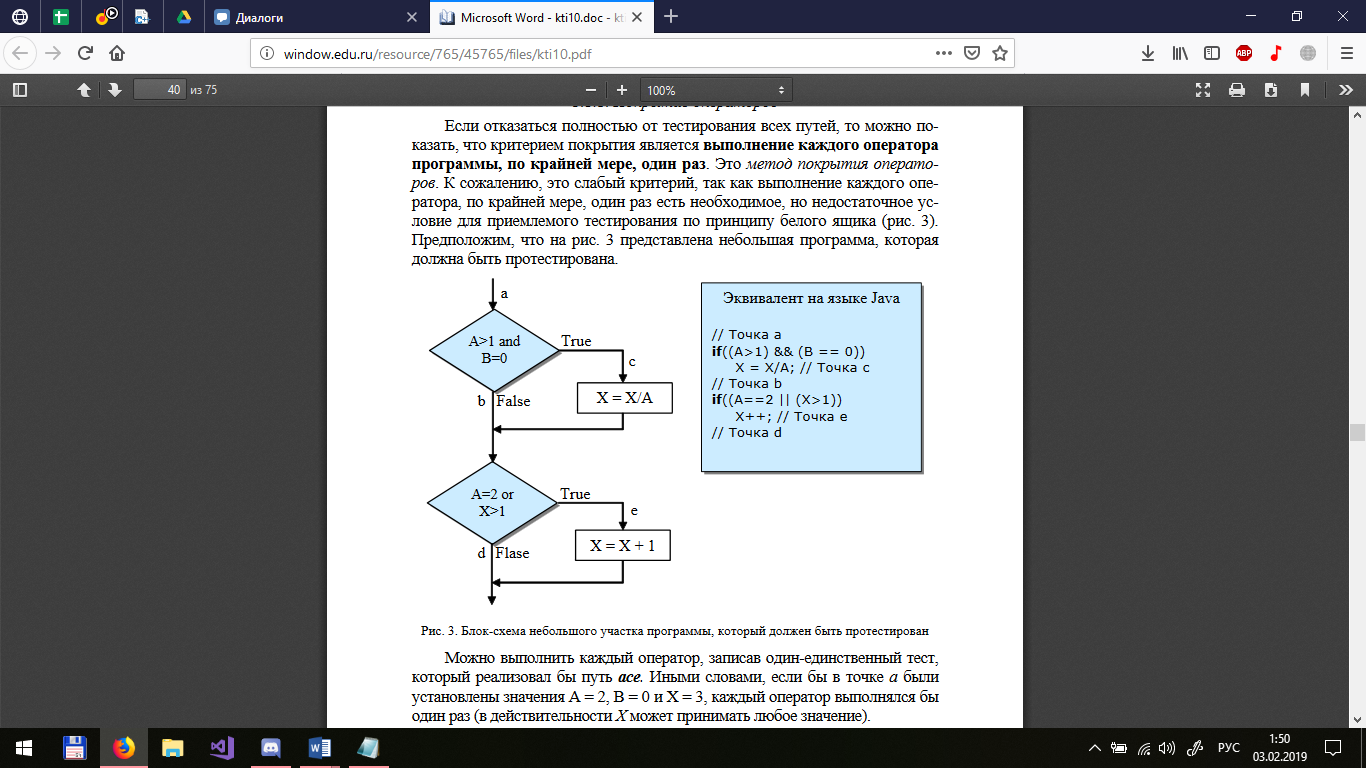


Рисунок 1. Блок-схема небольшого участка программы, который должен быть протестирован

Можно выполнить каждый оператор, записав один-единственный тест, который реализовал бы путь асе. Иными словами, если бы в точке а были установлены значения А = 2, В = 0 и Х = 3, каждый оператор выполнялся бы один раз (в действительности Х может принимать любое значение).

К сожалению, этот критерий хуже, чем он кажется на первый взгляд.

Например, пусть первое решение записано как «или», а не как «и» (в первом условии вместо “&&” стоит “||”). Тогда при тестировании с помощью данного критерия эта ошибка не будет обнаружена. Пусть второе решение записано в программе как Х > 0 (во втором операторе условия), эта ошибка также не будет обнаружена. Кроме того, существует путь, в котором Х не изменяется (путь abd). Если здесь ошибка, то и она не будет обнаружена. Таким образом, критерий покрытия операторов является настолько слабым, что его обычно не используют.

## **Покрытие решений**

Более сильный критерий покрытия логики программы (и метод тестирования) известен как покрытие решений, или покрытие переходов. Согласно данному критерию должно быть записано достаточное число тестов, такое, что каждое решение на этих тестах примет значение истина и ложь по крайней мере один раз. Иными словами, каждое направление перехода должно быть реализовано по крайней мере один раз. Примерами операторов перехода или решений являются операторы while или if.

Можно показать, что покрытие решений обычно удовлетворяет критерию покрытия операторов. Поскольку каждый оператор лежит на некотором пути, исходящем либо из оператора перехода, либо из точки входа программы, при выполнении каждого направления перехода каждый оператор должен быть выполнен. Однако существует, по крайней мере, три исключения.

Первое – патологическая ситуация, когда программа не имеет решений.

Второе встречается в программах или подпрограммах с несколькими точками входа (например, в программах на языке Ассемблера), данный оператор может быть выполнен только в том случае, если выполнение программы начинается с соответствующей точки входа.

Третье исключение – операторы внутри switch-конструкций, выполнение каждого направления перехода не обязательно будет вызывать выполнение всех case-единнц.

Так как покрытие операторов считается необходимым условием, покрытие решений, которое представляется более сильным критерием, должно включать покрытие операторов. Следовательно, покрытие решений требует, чтобы каждое решение имело результатом значения истина и ложь и при этом каждый оператор выполнялся бы, по крайней мере, один раз. Альтернативный и более легкий способ выражения этого требования состоит в том, чтобы каждое решение имело результатом значения истина и ложь и что каждой точке входа (включая каждую case-единицу) должно быть передано управление при вызове программы, по крайней мере, один раз.

Изложенное выше предполагает только двузначные решения или переходы и должно быть модифицировано для программ, содержащих многозначные решения (как для case-единиц). Критерием для них является выполнение каждого возможного результата всех решений, по крайней мере, один раз и передача управления при вызове программы или подпрограммы каждой точке входа, по крайней мере, один раз.

В программе, представленной на рис. 1, покрытие решений может быть выполнено двумя тестами, покрывающими либо пути асе и abd, либо пути acd и abe. Если мы выбираем последнее альтернативное покрытие, то входами двух тестов являются A = 3, В = 0, Х = 3 и A = 2, В = 1, Х = 1.

## **Покрытие условий**

Покрытие решений – более сильный критерий, чем покрытие операторов, но и он имеет свои недостатки. Например, путь, где Х не изменяется (если выбрано первое альтернативное покрытие), будет проверен с вероятностью 50 %. Если во втором решении существует ошибка (например, Х < 1 вместо Х > 1), то ошибка не будет обнаружена двумя тестами предыдущего примера.

Лучшим критерием (и методом) по сравнению с предыдущим является покрытие условий. В этом случае записывают число тестов, достаточное для того, чтобы все возможные результаты каждого условия в решении выполнялись, по крайней мере, один раз. Поскольку, как и при покрытии решений, это покрытие не всегда приводит к выполнению каждого оператора, к критерию требуется дополнение, которое заключается в том, что каждой точке входа в программу или подпрограмму, а также switch-единицам должно быть передано управление при вызове, по крайней мере, один раз.

Программа на рис. 1 имеет четыре условия: A > 1, B = 0, A = 2 и Х > 1.

Следовательно, требуется достаточное число тестов, такое, чтобы реализовать ситуации, где A > 1, A ≤ l, B = 0 и В ≠ 0 в точке а и A = 2, A ≠ 2, Х > 1 и Х ≤ 1 в точке b. Тесты, удовлетворяющие критерию покрытия условий, и соответствующие им пути:

1. A = 2, B = 0, Х = 4 асе.

2. A = 1, В = 1, Х = 1 abd.

Заметим, что, хотя аналогичное число тестов для этого примера уже было создано, покрытие условий обычно лучше покрытия решений, поскольку оно может (но не всегда) вызвать выполнение решений в условиях, не реализуемых при покрытии решений. Хотя применение критерия покрытия условий на первый взгляд удовлетворяет критерию покрытия решений, это не всегда так. Если тестируется решение if(A && B), то при критерии покрытия условий требовались бы два теста – А есть истина, В есть ложь и А есть ложь, В есть истина. Но в этом случае не выполнялось бы тело условия. Тесты критерия покрытия условий для ранее рассмотренного примера покрывают результаты всех решений, но это только случайное совпадение. Например, два альтернативных теста:

1. A = 1, В = 0, Х = 3.

2. A = 2, B = 1, Х = 1

покрывают результаты всех условий, но только два из четырех результатов решений (они оба покрывают путь аbе и, следовательно, не выполняют результат истина первого решения и результат ложь второго решения).

## **Покрытие решений/условий**

Очевидным следствием из этой дилеммы является критерий, названный покрытием решений/условий. Он требует такого достаточного набора тестов, чтобы все возможные результаты каждого условия в решении, все результаты каждого решения выполнялись, по крайней мере, один раз и каждой точке входа передавалось управление, по крайней мере, один раз.

Недостатком критерия покрытия решений/условий является невозможность его применения для выполнения всех результатов всех условий; часто подобное выполнение имеет место вследствие того, что определенные условия скрыты другими условиями. В качестве примера рассмотрим приведенную на рис. 2 схему передач управления в коде, генерируемым компилятором языка, программы рис. 1.

Многоусловные решения исходной программы здесь разбиты на отдельные решения и переходы, поскольку большинство компьютеров не имеет команд, реализующих решения со многими исходами. Наиболее полное покрытие тестами в этом случае осуществляется таким образом, чтобы выполнялись все возможные результаты каждого простого решения.

Два предыдущих теста критерия покрытия решений не выполняют этого, они недостаточны для выполнения результата ложь решения H и результата истина решения K. Набор тестов для критерия покрытия условий такой программы также является неполным; два теста (которые случайно удовлетворяют также и критерию покрытия решений/условий) не вызывают выполнения результата ложь решения I и результата истина решения K.

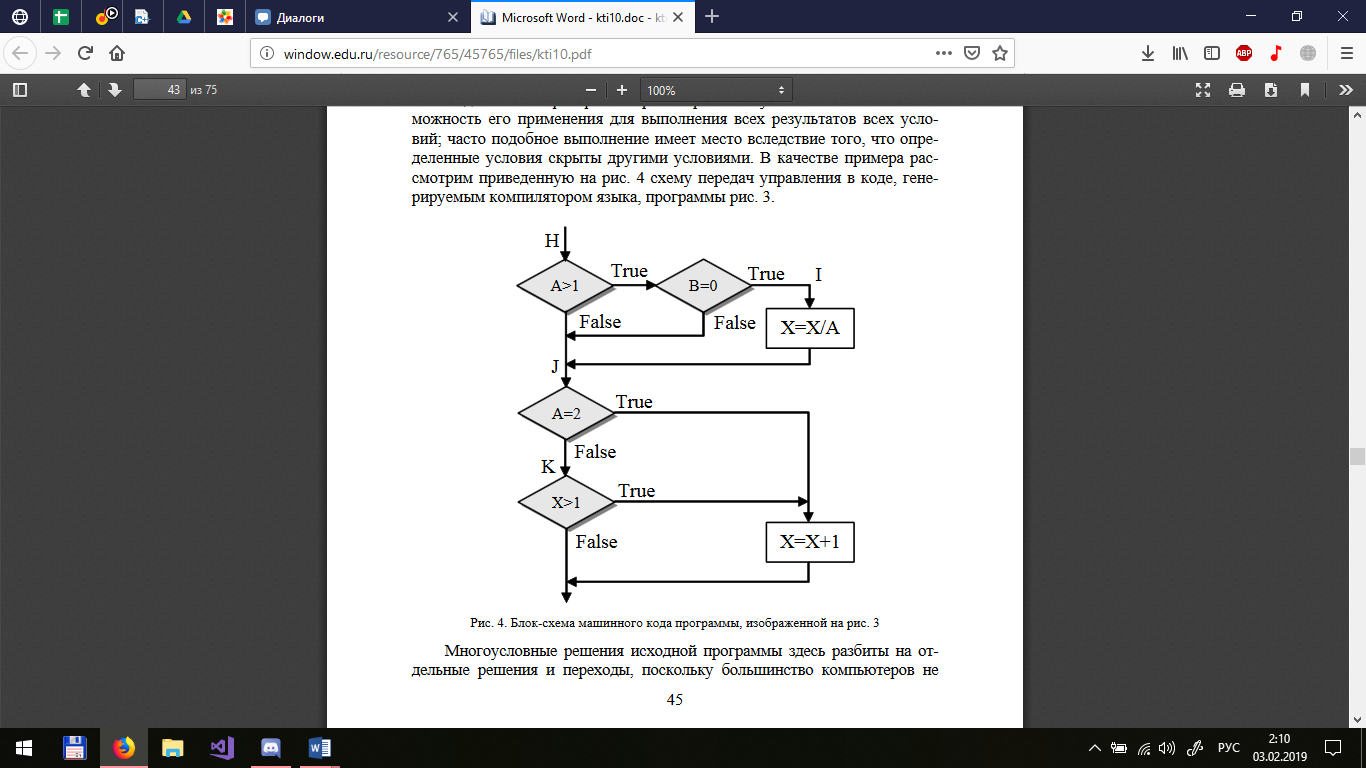


Рисунок 2.Блок-схема машинного кода программы, изображенной на рис. 1

Причина этого заключается в том, что, как показано на рис. 2, результаты условий в выражениях И и ИЛИ могут скрывать и блокировать действие других условий. Например, если условие и есть ложь, то никакое из последующих условий в выражении не будет выполнено. Аналогично если условие или есть истина, то никакое из последующих условий не будет выполнено. Следовательно, критерии покрытия условий и покрытия решений/условий недостаточно чувствительны к ошибкам в логических выражениях.

## **Комбинаторное покрытие условий**

Критерием, который решает эти и некоторые другие проблемы, является комбинаторное покрытие условий. Он требует создания такого числа тестов, чтобы все возможные комбинации результатов условия в каждом решении и все точки входа выполнялись, по крайней мере, один раз. По этому критерию для программы на рис. 3 должны быть покрыты тестами следующие восемь комбинаций:

1. А > 1, B = 0.

2. A > 1, В ≠ 0.

3. A ≤ 1, В = 0.

4. A ≤ l, В ≠ 0.

5. A = 2, X > 1.

6. A = 2, X ≤ l.

7. А ≠ 2, Х > 1.

8. A ≠ 2, X ≤ l.

Заметим, что комбинации 5–8 представляют собой значения второго оператора if. Поскольку Х может быть изменено до выполнения этого оператора, значения, необходимые для его проверки, следует восстановить, исходя из логики программы с тем, чтобы найти соответствующие входные значения.

Для того чтобы протестировать эти комбинации, необязательно использовать все восемь тестов. Фактически они могут быть покрыты четырьмя тестами. Приведем входные значения тестов и комбинации, которые они покрывают:

A = 2, B = 0, X = 4 покрывает 1, 5;

A = 2, В = 1, Х = 1 покрывает 2, 6;

A = 1, B = 0, Х = 2 покрывает 3, 7;

A = 1, B = 1, Х = 1 покрывает 4, 8.

То, что четырем тестам соответствуют четыре различных пути на рис. 1, является случайным совпадением. На самом деле представленные выше тесты не покрывают всех путей, они пропускают путь acd. Например, требуется восемь тестов для тестирования следующей программы на рисунке 3, хотя она покрывается лишь двумя путями. В случае циклов число тестов для удовлетворения критерию комбинаторного покрытия условий обычно больше, чем число путей.

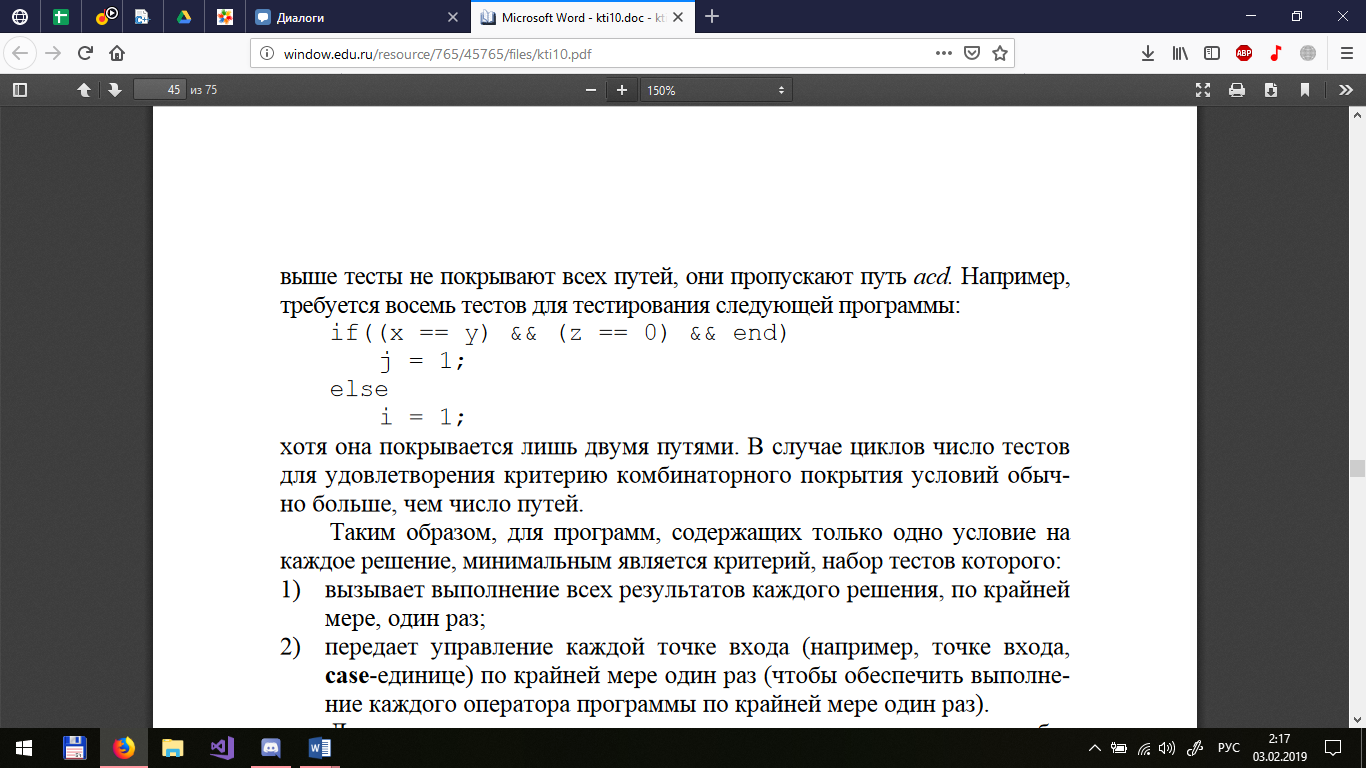


Рисунок 3

Таким образом, для программ, содержащих только одно условие на каждое решение, минимальным является критерий, набор тестов которого:

1) вызывает выполнение всех результатов каждого решения, по крайней мере, один раз;

2) передает управление каждой точке входа (например, точке входа, case-единице) по крайней мере один раз (чтобы обеспечить выполнение каждого оператора программы по крайней мере один раз). (Йодан, 1979)

Для программ, содержащих решения, каждое из которых имеет более одного условия, минимальный критерий состоит из набора тестов, вызывающих выполнение всех возможных комбинаций результатов условий в каждом решении и передающих управление каждой точке входа программы, по крайней мере, один раз. Слово «возможных» употреблено здесь потому, что некоторые комбинации условий могут быть нереализуемыми, например, в выражении (a>2) && (a<10) могут быть реализованы только три комбинации условий.

## **Преимущества и недостатки**

**Преимущества:**

– тестирование может производиться на ранних этапах: нет необходимости ждать создания пользовательского интерфейса;

– можно провести более тщательное тестирование, с покрытием большого количества путей выполнения программы.

**Недостатки:**

– для выполнения тестирования белого ящика необходимо большое количество специальных знаний

– при использовании автоматизации тестирования на этом уровне, поддержка тестовых скриптов может оказаться достаточно накладной, если программа часто изменяется.

# **Методологии “серый ящик”**

При тестировании по стратегии серого ящика руководствуются   
не только спецификацией, но и ключевыми элементами проектирования. Тестируется как функционал, так и ожидаемое поведение программы.  
Техники тестирования серого ящика :   
- регрессионное тестирование;  
- шаблонное тестирование;  
- ортогональный массив.

## **Регрессионное тестирование**

**Регрессионное тестирование** – это набор тестов, направленных на обнаружение дефектов в уже протестированных участках приложения. Делается это совсем не для того, чтобы окончательно убедиться в отсутствии багов, а для поиска и исправления регрессионных ошибок. Регрессионные ошибки – те же баги, но появляются они не при написании программы, а при добавлении в существующий билд нового участка программы или исправлении других багов, что и стало причиной возникновения новых дефектов в уже протестированном продукте.

Таким образом, мы можем сказать, что цель регрессионного тестирования – убедиться, что исправление одних багов не стало причиной возникновения других и что обновление билда не создало новых дефектов в уже проверенном коде.

**Ортогональный массив**

Стратегия тестирования ортогональных массивов (или "OATS") - это подход выбора теста, который выбирает очень разнообразный набор тестовых сценариев, чтобы найти как можно больше ошибок в как можно меньших тестах. Это мощный подход к тестовому дизайну, который набирает популярность, поскольку он доказал, что повышает эффективность и эффективность тестирования во многих различных типах тестовых контекстов.

Используя OATS, тестировщики могут стратегически идентифицировать управляемое количество высокоприоритетных тестов в ситуациях, когда на выбор могут быть тысячи, миллионы, миллиарды возможных перестановок.

**Шаблонное тестирование**

Шаблонное тестированиеявляется инструментом для автоматизации регрессионного функционального тестирования конфигураций, т.е. выполнения некоторых действий (выполнение и проверка результатов произвольного запроса и т.п.) в базе с помощью создания и выполнения шаблонов действий и проверки результата.

## **Преимущества и недостатки**

**Преимущества:**

- Тестирование обеспечивает комбинированные преимущества как белого ящика так и тестирование черного ящика

- Оно основано на функциональной спецификации, UML, диаграмм базы данных или архитектурной точки зрения

Дополнительное преимущество тестирования серого ящика является то, что оно поддерживает границу между независимыми тестерами и разработчиками.

**Недостатки:**

- При испытании серого ящика, полное тестирование белого ящика не может быть сделано из-за недоступных исходного кода .

-Трудно связать дефекты, когда проводится тестирование серого ящика для распределенной системы.

# **Заключение**

Тестирование белого ящика смещает акцент с вопроса "что должен делать код" на "что фактически делает код". Иными словами, вместо использования более высокого уровня абстракции, формирования тестов на основе спецификации, используется точно тот же уровень абстракции, что и при реализации кода. Мы можем получить хорошие результаты в плане покрытия кода, но при этом такое тестирование имеет смысл в ограниченном наборе случаев.

Метод «серого ящика» помогает в случаях когда нет возможности использовать «белый ящик» или необходимо более полное покрытие по сравнению с «черным ящиком». Используя этот метод, тестировщики получают доступ к проектной документации и могут подготовить и создать более точные и полные тест-кейсы и сценарии тестирования. Наибольшая эффективность применения «серого ящика» достигается при тестировании web-приложений, web-сервисов, безопасности, GUI, а также для функционального тестирования.

# **Список литературы**

**Искусство тестирования программ [Книга] / авт. Г. Майерс. - [б.м.] : Мир, 1982. - стр. 176.**

**Методы тестирования программного обеспечения [Книга] / авт. Степанченко И. В.. - [б.м.] : РПК «Политехник», 2006. - стр. 43.**

**Надежность программного обеспечения [Книга] / авт. Г. Майерс. - [б.м.] : Мир, 1980..**

**Структурное проектирование и конструирование программ [Книга] / авт. Э. Йодан. - [б.м.] : Мир, 1979. - стр. 416.**