**Министерство образования Республики Беларусь**

**Белорусский государственный университет**

**Факультет прикладной математики и информатики**

**Реферат**

**Основы тестирования программного обеспечения. Разновидности тестирования. Тестирование программы как "черного ящика” или функциональное тестирование**

***Выполнил:***

студент 1 курса, 2 группы факультета

прикладной математики и

информатики БГУ специальности

“Информатика” очной формы

Обучения

**Бакатович Антон Олегович**

***Преподаватель:***

**Баранов Геннадий Аркадьевич**

Минск

2017

**СОДЕРЖАНИЕ**

[**ВВЕДЕНИЕ** 3](#_Toc480735610)

[**Основы тестирования программного обеспечения.** 3](#_Toc480735611)

[**Разновидности тестирования.** 3](#_Toc480735612)

[**Тестирование программы как "черного ящика” или функциональное тестирование.** 4](#_Toc480735613)

[**Эквивалентное разбиение.** 6](#_Toc480735614)

[**Анализ граничных значений** 7](#_Toc480735615)

[**Применение функциональных диаграмм.** 10](#_Toc480735616)

[**ЛИТЕРАТУРА** 16](#_Toc480735617)

# **ВВЕДЕНИЕ**

Тестирование – важная часть любой программы контроля качества, а зачастую и единственная. Первые программные системы разрабатывались в рамках программ научных исследований или программ для нужд министерств обороны. Тестирование таких продуктов проводилось строго формализовано с записью всех тестовых процедур, тестовых данных, полученных результатов. Тестирование выделялось в отдельный процесс, который начинался после завершения кодирования, но при этом, как правило, выполнялось тем же персоналом. Систематичный подход к тестированию, выполняемому разработчика, позволяет находить максимальное число дефектов всех типов при минимуме усилий.

# **Основы тестирования программного обеспечения**

Тестирование – самая популярная методика повышения качества, подкреплённая многими исследованиями и богатым опытом разработки коммерческих приложений. Тестирование имеет следующие цели:

* Продемонстрировать разработчикам и заказчикам, что программа соответствует требованиям.
* Выявить ситуации, в которых поведение программы является неправильным, нежелательным или не соответствующим спецификации.

Тестирование требует, чтобы вы рассчитывали найти ошибки в своём коде. В противном случае вы, вероятно, на самом деле их не найдете, но это будет всего лишь пророчеством. Если вы запускаете программу в надежде, что она не содержит ошибок, их будет слишком легко не заметить. В исследовании, которое уже стало классическим, Майерс попросил группу опытных программистов протестировать программу, содержащую 15 известных дефектов. В среднем программисты нашли лишь 5 из 15 ошибок. Лучший программист обнаружил только 9. Главной причиной неэффективного обнаружения ошибок было недостаточно внимательное изучение ошибочных выходных данных. Ошибки были видны, но программисты их не заметили.

# **Разновидности тестирования**

Существует множество видов тестирования: одни обычно выполняют сами разработчики, а другие – специализированные группы. Виды тестирования перечислены ниже.

* *Блочным тестированием* называют тестирование полного класса, метода или небольшого приложения, написанного одним программистом или группой, выполняемое отдельно от прочих частей системы.
* *Тестирование компонента –* это тестирование класса, пакета, небольшого приложения или другого элемента системы, разработанного несколькими программистами или группами, выполняемое в изоляции от остальных частей системы.
* *Интеграционное тестирование* — это совместное выполнение двух или более классов, пакетов, компонентов или подсистем, созданных несколькими программистами или группами. Этот вид тестирования обычно начинают проводить, как только созданы два класса, которые можно протестировать, и продолжают до завершения работы над системой.
* Регрессивным тестированием называют повторное выполнение тестов, направленное на обнаружение дефектов в программе, уже прошедшей этот набор тестов.
* *Тестирование системы* — это выполнение ПО в его окончательной конфигурации, интегрированного с другими программными и аппаратными системами. Предметом тестирования в этом случае являются безопасность, производительность, утечка ресурсов, проблемы синхронизации и прочие аспекты, которые невозможно протестировать на более низких уровнях интеграции.

Тестирование, выполняемое разработчика, обычно включает три первых вида тестирования из приведённого списка, но иногда может включать также регрессивное тестирование и тестирование системы. Многие другие виды тестирования обычно выполняют не разработчики, а специализированный персонал: в качестве примеров можно привести бета-тестирование, тестирование системы на предмет одобрения заказчиком, тестирование производительности, тестирование конфигурации, платформенное тестирование, тестирование в стрессовом режиме, тестирование удобства использования и т. д.

# **Тестирование программы как "черного ящика” или функциональное тестирование**

Тестирование чёрного ящика— стратегия тестирования функционального поведения объекта (программы, системы) с точки зрения внешнего мира, при котором не используется знание о внутреннем устройстве тестируемого объекта. Под стратегией понимаются систематические методы отбора и создания тестов для тестового набора. Стратегия поведенческого теста исходит из технических требований и их спецификаций.

Под «чёрным ящиком» понимается объект исследования, внутреннее устройство которого неизвестно. В кибернетике оно позволяет изучать поведение систем, то есть их реакций на разнообразные внешние воздействия и в то же время абстрагироваться от их внутреннего устройства.

Манипулируя только лишь со входами и выходами, можно проводить определенные исследования. На практике всегда возникает вопрос, насколько гомоморфизм «чёрного» ящика отражает адекватность его изучаемой модели, то есть как полно в модели отражаются основные свойства оригинала.

Описание любой системы управления во времени характеризуется картиной последовательности её состояний в процессе движения к стоящей перед нею цели. Преобразование в системе управления может быть либо взаимно-однозначным и тогда оно называется изоморфным, либо только однозначным, в одну сторону. В таком случае преобразование называют гомоморфным.

«Чёрный» ящик представляет собой сложную гомоморфную модель кибернетической системы, в которой соблюдается разнообразие. Он только тогда является удовлетворительной моделью системы, когда содержит такое количество информации, которое отражает разнообразие системы. Можно предположить, что чем большее число возмущений действует на входы модели системы, тем большее разнообразие должен иметь регулятор.

В настоящее время известны два вида «чёрных» ящиков. К первому виду относят любой «чёрный» ящик, который может рассматриваться как автомат, называемый конечным или бесконечным. Поведение таких «чёрных» ящиков известно. Ко второму виду относятся такие «чёрные» ящики, поведение которых может быть наблюдаемо только в эксперименте. В таком случае в явной или неявной форме высказывается гипотеза о предсказуемости поведения «чёрного» ящика в вероятностном смысле. Без предварительной гипотезы невозможно любое обобщение, или, как говорят, невозможно сделать индуктивное заключение на основе экспериментов с «чёрным» ящиком. Для обозначения модели «чёрного» ящика Н. Винером предложено понятие «белого» ящика. «Белый» ящик состоит из известных компонентов, то есть известных X, Y, δ, λ. Его содержимое специально подбирается для реализации той же зависимости выхода от входа, что и у соответствующего «чёрного» ящика. В процессе проводимых исследований и при обобщениях, выдвижении гипотез и установления закономерностей возникает необходимость корректировки организации «белого» ящика и смены моделей.

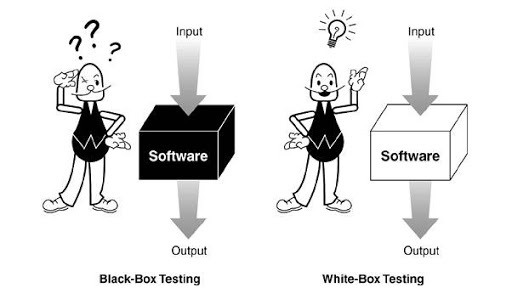


Рисунок 1 – чёрный и белый ящики

# **Эквивалентное разбиение**

Хороший текст должен покрывать широкий диапазон входных данных. Если два текста приводят к обнаружению одних и тех же ошибок, вам нужен лишь один из них. «разделения на классы эквивалентности» формализует эту идею и помогает уменьшить число нужных тестов.

Эквивалентное разбиение – это метод тестирования «черного ящика». Идея тестирования по методу разбиения классов эквивалентности состоит в том, чтобы исключить набор входных данных, которые заставляют систему вести себя одинаково и давать одинаковый результат при тестировании программы.

Разбиение на разделы является обычной методикой тестирования с использованием черного ящика и нацелено на сокращение числа избыточных тестовых сценариев, устраняя те, которые генерируют один и тот же результат, и не обязательно обнаруживают дефекты в функциональности программы.

Поскольку целью тестирования является обнаружение дефектов, то успешный тестовый сценарий – тот, который обнаруживает дефект.

Процесс тестирования по методу эквивалентного разбиения включает в себя идентификацию набора данных как условий ввода, которые дают одинаковый результат при выполнении программы и классифицируют их как набор эквивалентных данных (поскольку они заставляют программу вести себя одинаково и генерировать тот же результат) и происходит разбиение их на группы из другого эквивалентного набора данных.

Рассмотрим пример. Можно ожидать, что приложение, принимающее целочисленные значения (целочисленные числа) от -1,000 до +1,000, сможет обрабатывать отрицательные целые числа, ноль и положительные целые числа. Таким образом, набор входных значений можно разделить на три раздела:

От -1 000 до -1, 0 и от 1 до 1000

Более того, ожидается, что система будет вести себя одинаково для значений внутри каждого раздела. То есть способ, которым система обрабатывает -537, будет такой же, как -9. Аналогично, положительные целые числа 5 и 729 будут обрабатываться системой одинаково. В этом конкретном примере значение 0 является разделом с одним значением. Обычно хорошей практикой является специальный сценарий с нулевым числом.

Важно отметить, что эта методика применима не только к цифрам. Этот метод может применяться к любому набору данных, который можно рассматривать как эквивалент.

Размышление о разделении на классы эквивалентности не скажет вам много нового о программе, если вы уже покрыли ее базисным тестированием и тестированием, основанным на потоках данных. Однако оно очень полезно, если вы смотрите на программу «извне» (с точки зрения спецификации, а не исходного кода) или если данные сложны, а эта сложность плохо отражена в логике программы.

# **Анализ граничных значений**

Анализ граничных значений – эта техника проверки ошибок на границе классов эквивалентности.

Если техника анализа классов эквивалентности ориентирована на тестовое покрытие, то эта техника основана на рисках. Эта техника начинается с идеи о том, что программа может сломаться в области граничных значений.

Причины, по которым считается, что с граничными значениями связаны серьёзные риски:

* Давно замечено, что при разработке большое число проблем возникает на границах входных данных.
* Даже если эквивалентные классы найдены правильно, то граничные значения могут быть ошибочно отнесены к другому классу.

Эта техника на первый взгляд простая. Однако эффективное применение этой техники зависит от способности правильно выделить классы эквивалентности и затем выбрать тесты для проверки границ этих классов.

Цель этой техники сформулировать несложно - найти ошибки, связанные с граничными значениями.

Примерный алгоритм использования техники анализа граничных значений:

1. Выделить классы эквивалентности.
2. Определить граничные значения этих классов.
3. Нужно понять, к какому классу будет относиться каждая граница.
4. Для каждой границы нам нужно провести тесты по проверке значения до границы, на границе, и сразу после границы.

Рассмотрим пример с использованием анализа граничных значений.

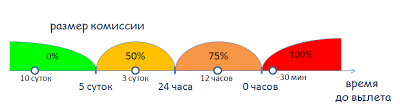


Рисунок 2 – пример использования граничных значений

Предположим, что размер комиссии зависит от времени до вылета, когда совершена отмена:

* За 5 суток до вылета комиссия составляет 0%.
* Меньше 5 суток, но больше 24 часов – 50%.
* Меньше 24 часов, но до вылета – 75%.
* После вылета – 100%.

Воспользуемся алгоритмом, приведённым выше

1. Выделим классы эквивалентности:

* Время до вылета больше 5 суток.
* Время до вылета больше 24 часов, но меньше 5 суток.
* Время до вылета больше 0 часов, но меньше 24 часов
* Вылет уже состоялся.

1. Определим границы:

* 5 суток.
* 24 часа.
* 0 часов.

1. Определим, к какому классу относятся границы:

* 5 суток – к 2-му классу.
* 24 часа – к 2-му классу.
* 0 часов – к 4-му классу.

1. Протестируем значения на границах, до и после них:

* Отменим бронь за 5 суток + 1 секунда до вылета (или просто постараемся выполнить бронь как можно ближе к границе, но слева от неё) и проверим, что комиссия равна 0%.
* Отменим бронь ровно за 5 суток до вылета и проверим, что комиссия равна 50%.
* Отменим бронь за 5 суток – 1 секунда до вылета и проверим, что комиссия равна 50%.
* Отменим бронь за 24 часа + 1 секунда до вылета и проверим, что комиссия равна 50%.
* Отменим бронь ровно за 24 часа до вылета и проверим, что комиссия равна 50%.
* Отменим бронь за 24 часа – 1 секунда до вылета и проверим, что комиссия равна 75%.
* Отменим бронь за 1 секунду до вылета и проверим, что комиссия равна 75%.
* Отменим бронь ровно во время вылета и проверим, что комиссия равна 100%.
* Отменим бронь спустя 1 секунду после вылета и проверим, что комиссия равна 100%.

Мы получили 9 тестов, по 3 теста на каждую границу. Если суммировать тесты, необходимые для проверки классов эквивалентности и граничных значений, то получим 13 тестов.

Иногда рекомендуется использовать классы эквивалентности и граничные условия вместе по следующим соображениям:

1. Техника анализа классов эквивалентности говорит о том, что должно выбираться минимум одно значение из каждого класса.
2. Так как граница обычно относится к какому-то классу, то можно

использовать её как представителя этого класса.

1. Экономия определённого количества тестов.

Можно выделить следующее преимущество техники анализа граничных значений:

Эта техника добавляет в технику анализа классов эквивалентности ориентированность на конкретный тип ошибок. То есть, техника анализа классов эквивалентности просто говорит нам о том, что нужно разбить все тесты на классы и провести тестирование всех классов. А техника граничных значений ориентирована на обнаружение конкретной проблемы – возникновения ошибок на границах классов эквивалентности.

Но, как и для техники анализа классов эквивалентности, эффективность техники анализа граничных значений зависит от правильности ее использования. Должны прилагаться усилия, чтобы правильно определить классы эквивалентности и их границы. Если отнестись к этому поверхностно, то возникнет риск пропустить ошибки.

# **Применение функциональных диаграмм**

Одним из недостатков анализа граничных значений и эквивалентного разбиения является то, что они не исследуют комбинаций входных условий.

Метод функциональных диаграмм или диаграмм причинно-следственных связей помогает систематически выбирать тесты с высокой результативностью. Он дает полезный побочный эффект, так как позволяет обнаруживать неполноту и неоднозначность исходных спецификаций.

Функциональная диаграмма представляет собой формальный язык, на который транслируется спецификация, написанная на естественном языке. Диаграмме можно сопоставить цифровую логическую цепь, но для ее описания используется более простая нотация, чем обычная форма записи, принятая в электронике. Для уяснения метода функциональных диаграмм вовсе не обязательно знание электроники, но желательно понимание булевской логики (т.е. логических операторов и, или и не). Построение тестов этим методом осуществляется в несколько этапов:

1. Спецификация разбивается на «рабочие» участки. Это связано с тем, что функциональные диаграммы становятся слишком громоздкими при применении данного метода к большим спецификациям. Например, когда тестируется система разделения времени, рабочим участком может быть спецификация отдельной команды. При тестировании компилятора в качестве рабочего участка можно рассматривать каждый отдельный оператор языка программирования.
2. В спецификации определяются причины и следствия. Причина есть отдельное входное условие или класс эквивалентности входных условий. Следствие есть выходное условие или преобразование системы (остаточное действие, которое входное условие оказывает на состояние программы или системы). Например, если сообщение программе приводит к обновлению основного файла, то изменение в нем и является преобразованием системы; подтверждающее сообщение было бы выходным условием. Причины и следствие определяются путем последовательного (слово за словом) чтения спецификации. При этом выделяются слова или фразы, которые описывают причины и следствия. Каждым причине и следствию приписывается отдельный номер.
3. Анализируется семантическое содержание спецификации, которая преобразуется в булевский граф, связывающий причины и следствия. Это и есть функциональная диаграмма.
4. Диаграмма снабжается примечаниями, задающими ограничения и описывающими комбинации причин и (или) следствий, которые являются невозможными из-за синтаксических или внешних ограничений.
5. Путем методического прослеживания состояний усло­вий диаграммы она преобразуется в таблицу решений с ограниченными входами. Каждый столбец таблицы решений соответствует тесту.
6. Столбцы таблицы решений преобразуются в тесты.

Процедура генерации таблицы решений заключается в следующем:

* Выбрать некоторое следствие, которое должно быть в состоянии.
* Найти все комбинации причин (с учётом ограничений), которые установят это следствие в 1, прокладывая из этого следствия обратную трассу через диаграмму.
* Построить столбец в таблице решений для каждой комбинации причин.
* Для каждой комбинации причин определить состояния всех других следствий и поместить их в соответствующий столбец таблицы решений.

При выполнении этого шага необходимо руководствоваться следующими положениями:

* Если обратная трасса прокладывается через узел **или**, выход которого должен принимать значение 1, то одновременно не следует устанавливать в 1 более одного входа в этот узел. **Цель данного правила**– избежать пропуска определенных ошибок из-за того, что одна причина маскируется другой.
* Если обратная трасса прокладывается через узел **или**, выход которого должен принимать значение 0, то все комбинации входов, приводящие выход в 0, должны быть в конечном счете перечислены. Однако, когда исследуется ситуация, где один вход есть 0, а один или более других входов есть 1, не обязательно перечислять все условия, при которых остальные входы могут быть 1.
* Если обратная трасса прокладывается через узел и, выход которого должен принимать значение 0, то необходимо указать лишь одно условие, согласно которому все входы являются нулями.

Узел диаграммы может находиться в двух состояниях – 0 или 1. 0 обозначает состояние “отсутствует”, а 1 – “присутствует”.

Для представления функциональных диаграмм используются следующие базовые символы:

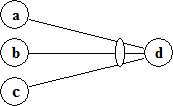


Рисунок 3 – функция или

Функция или устанавливает, что если a, или b, или c есть 1, то d есть 1. В противном случае d есть 0.

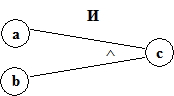


Рисунок 4 – функция или

Функция и устанавливает, что если a, и b есть 1, то и c есть 1. В противном случае c есть 0.

Рассмотрим пример:

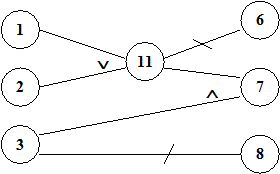


Рисунок 4 – диаграмма спецификаций

Файл обновляется, если символ в колонке 1 является буквой “A” или “B”, а символ в колонке 2 – цифра. Если первый символ ошибочный, то выдаётся сообщение X1, а если второй символ не является цифрой – сообщение X2.

Причины:

1. Символ “A” в колонке 1.
2. Символ “B” в колонке 1.
3. Цифра в колонке 2.

Следствия:

7. Файл обновляется.

8. Выдаётся сообщение X1.

9. Выдаётся сообщение X2.

Для изображения невозможных комбинаций причин используется следующие логические ограничения:

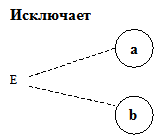


Рисунок 5 – ограничение E

Ограничение E устанавливает, что E должно быть истинным, если хотя бы одна из причин – a или b – принимает значение 1. Стоит обратить внимание, что a и b не могут принимать значения 1 одновременно.

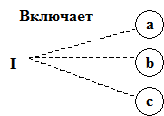


Рисунок 6 – ограничение I

Ограничение I устанавливает, что по крайней мере одна из величин a, b или c всегда должна быть равной 1 (a, b и c не могут принимать значение 0 одновременно).

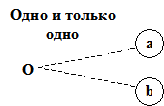


Рисунок 7 – ограничение O

Ограничение O устанавливает, что одна и только одна из величин a или b должна быть равной 1.

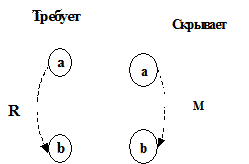


Рисунок 8 – ограничения R и M

Ограничение R устанавливает, что если a принимает значение 1, то b должна принимать значение 1, то есть невозможно, чтобы a было равно 1, а b - 0.

Ограничение M устанавливает, что если следствие a имеет значение, то следствие b должно принять значение 0.

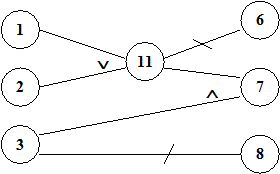


Рисунок 8 – итоговая диаграмма

Для рассмотренного выше примера невозможно, чтобы причины 1 и 2 присутствовали одновременно, но возможно, чтобы присутствовала одна из них. Следовательно, они связаны ограничением E.

Преобразуем полученную функциональную диаграмму в таблицу решений. Выберем следствие 7 (файл обновляется). Следствие 7 имеет место, если узлы 3 и 11 есть 1. А узел 11 есть 1, если одна из причин 1 или 2 имеет значение 1. Таким образом, возможны следующие состояния узлов 1 – 3:

1 0 1 и 0 1 0.

Следствие 6 имеет место, если значение узла 11 есть 0 (узлы 1 и 2 равны 0).

Следствие 8 имеет место, если значение узла 3 есть 0.

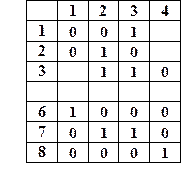


Рисунок 9 –таблица решений

Столбец 1 представляет условие, где следствие 6 есть 1, столбцы 2, 3 – следствие 7 есть 1, а столбец 4 соответствует условию, для которого следствие 8 есть 1.

Пробелы в таблице решений представляют собой “безразличные” ситуации.

Преобразуем таблицу решений в набор тестов:

1. 1 1
2. A 2
3. B 2
4. A A

# **ЛИТЕРАТУРА**

1. С. Макконнелл. Совершенный код. — СПб: «Питер», 2005. — 896 с.
2. Тестирование объектно-ориентированного программного обеспечения, К: Диасофт, 2002. - 432с.
3. Тестирование программ: цели и особенности инструментальной поддержки // Программное обеспечение ЭВМ / АН БССР. Институт математики. Минск, 1994. - Вып. 100 - с. 19 - 43
4. Б. Бейзер. Тестирование черного ящика. — СПб: «Питер», 2005. — 318 с.