**БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ**

**Факультет прикладной математики и информатики**

**Кафедра информационных систем управления**

### 

**РЕФЕРАТ**

**на тему “Модульное и интеграционное тестирование”**

**студента 1 курса 2 группы**

**Выполнил:**

студент 1 курса, 2 группы факультета прикладной математики и информатики БГУ специальности “Информатика” очной формы обучения

**Дорошев Игорь Викторович**

**Преподаватель**

Баранов Геннадий Аркадьевич

Минск

2017

**Содержание**

[**Введение** 3](#_Toc480740253)

[**Модульное тестирование** 4](#_Toc480740254)

[**Интеграционное тестирование** 7](#_Toc480740255)

[ **Восходящее тестирование:** 8](#_Toc480740256)

[ **Монолитное тестирование:** 8](#_Toc480740257)

[ **Нисходящее тестирование:** 9](#_Toc480740258)

[**Особенности тестирования для ООП** 10](#_Toc480740259)

[**Особенности индустриального тестирования** 11](#_Toc480740260)

[**Тестовый цикл** 11](#_Toc480740261)

[**Тестовый план** 12](#_Toc480740262)

[**Регрессионное тестирование** 13](#_Toc480740263)

[**Преимущества и недостатки регрессионного тестирования** 14](#_Toc480740264)

[**Список литературы** 16](#_Toc480740265)

[**Ресурсы** 16](#_Toc480740266)

# **Введение**

*Тестирование* - важная часть процесса разработки любой программы. Трудно себе представить процесс разработки коммерческой программы без стадии тестирования. Тестирование является одним из видов контроля качества программного обеспечения, состоит из нескольких этапов и позволяет найти ошибки, допущенные программистами на стадии конструирования программы, ведущие к ее некорректной работе. Отсутствие надлежащего тестирования ведёт к обнаружению таких ошибок конечными пользователями, что крайне нежелательно.

Выделяют несколько этапов тестирования, и рекомендуется выполнять каждый из них для обнаружения как можно большего числа ошибок, допущенных на стадии конструирования программного продукта. Выполнение каждого из отдельных этапов тестирования ведёт к обнаружению менее 50% ошибок. Комбинация различных этапов тестирования часто приводит к обнаружению менее 60% ошибок.

На практике используются следующие этапы тестирования:

* модульное тестирование
* интеграционное тестирование
* системное тестирование
  + альфа-тестирование
  + Бета-тестирование

Этапы отличаются не только выполняемыми действиями, но и стадией разработки программы, на которой они используются. Тестирование может выполняться как на стадии разработки (конструирования), так и на стадии использования. Согласно исследованиям, выявление и исправление ошибок на стадии конструирования позволяет сократить стоимость работ на 25%. Во время конструирования программы выполняются модульный и интеграционный этапы тестирования. Рассмотрим их подробнее.

# **Модульное тестирование**

*Модульное тестирование* используется для проверки отдельных блоков кода (методов, функций, процедур). Как правило, применяется к каждому нетривиальному блоку, чтобы убедиться в том, что эта часть кода написана корректно и не вносит ошибок в существующую программу. Цель такого тестирования - показать, что отдельные блоки программы являются работоспособными.

Процесс модульного тестирования заключается в том, чтобы изолировать небольшой фрагмент кода - тестируемый модуль - от остальной программы, написать код тестирующего блока и вызывать тестируемый блок, передавая ему необходимые параметры. После выполнения тестируемого блока, возвращенное им значение сравнивается с ожидаемым корректным значением для заданных входных параметров. На основе такого сравнения делается вывод, корректно ли отработал код. Каждый модуль необходимо тестировать таким образом отдельно, перед его интеграцией в систему.

Преимущества такого подхода заключаются в следующем:

* такое тестирование позволяет проводить рефакторинг, будучи уверенным, что модуль по-прежнему работает корректно. Это делает процесс изменения кода удобным и простым, поскольку легко проверить, работает ли модуль после внесённых изменений.
* является способом тестирования “снизу вверх”: сначала тестируются отдельные части программы, а затем, код целиком.
* обеспечение работоспособности отдельных компонент упрощает интеграцию.
* юнит-тесты можно рассматривать как часть документации: код тестирующего модуля можно использовать как пример применения тестируемого модуля.
* тестирование корректности архитектуры классов: модульное тестирование позволяет убедиться в корректности реализации класса (надлежащей инкапсуляцией компонент).
* модульное тестирование требует разбиения текста на модули, а значит, такой код легче обслуживать и использовать повторно.
* ускоряет процесс разработки за счет обеспечения корректности небольших модулей, составляющих целую программу.
* локализация ошибок играет большую роль: модульное тестирование позволяет найти модуль, содержащий ошибку - небольшой фрагмент кода, быстро её найти и исправить. Рекомендуется не удалять тестирующий код после исправления ошибки.
* тестирование отдельных компонент не приводит к разрастанию ошибочного кода в большие фрагменты, что обеспечивает простоту исправления допущенных ошибок.

Для корректного использования модульного тестирования необходимо хорошо разбираться в структуре программы, чтобы покрыть код тестами с минимальными затратами - тестирующий код не должен разрастаться слишком быстро, а его эффективность должна быть гарантирована корректностью и продуманностью используемых тестов. Рекомендуется использовать готовые инструменты и библиотеки, упрощающие процесс тестирования. Модульное тестирование позволяет легко отделить тестирующий код от тестируемого, а значит процесс тестирования становится более гибким, так как тестирующий код изолирован и его легко менять. Кроме того, отделение тестирующего кода от тестируемого гарантирует, что тестирующий код не нарушит инкапсуляции и не будет использовать закрытые данные, то есть процесс тестирования является лишь кодируемым вариантом использования программы.

Автоматизированный процесс тестирования позволяет определить не только корректность модуля, но и его производительность - провести нагрузочное тестирование модуля просто запрограммировав различные его вызовы в большом количестве, основываясь на анализе его “узких мест” и программными средствами измерить время выполнения модуля. Также, кроме нагрузочного тестирования и использования запланированного набора тестов, можно использовать генератор случайных тестов.

При использовании модульного тестирования рекомендуется сохранять тесты и их результаты отдельно, чтобы исключить необходимость создавать уже использовавшиеся тесты и запускать повторно тесты на протестированном коде.

Однако, модульное тестирование также имеет свои недостатки:

* необходимость покрывать код большим количеством тестов
* необходимость учитывать различные состояния переменных тестируемого модуля: например, если используется булевая переменная, необходимо учитывать поведение метода для каждого из возможных её состояний - True и False, что ведёт к появлению большого количества тестов.
* часто случаются ситуации, когда ожидаемое значение посчитать сложно, тогда использование модульного тестирования вызывает затруднения и не всегда ведёт к эффективному контролю качества программного продукта.
* тестирование каждого модуля по отдельности не отображает поведения нескольких модулей в связке, что означает недостаточность использования лишь модульного тестирования.
* выполнение большого количества модульных тестов может занять долгое время.
* модульное тестирование подразумевает определенную структуру программы и становится неприменимым при низкой общей культуре кода.
* при частом изменении модулей необходимо проводить тестирование заново
* за исключением простейших случаев, тестируемый код может взаимодействовать с другими компонентами системы, что требует их готовности: при наличии недописанных компонент, или заглушек модульное тестирование может оказаться неприменимым.

Несмотря на недостатки, модульное тестирование является большой и важной частью контроля качества программы, позволяя сократить время и расходы на исправление ошибок. Модульное тестирование также используется в так называемой “разработке через тестирование”, когда тесты пишутся до написания кода и обозначают задачи, поставленные перед модулем, а сам процесс разработки подразумевает написание кода, удовлетворяющего тестам. При надлежащем использовании, модульное тестирование обеспечит мощный механизм тестирования программы и позволит вовремя обнаружить ошибки, допущенные в коде.

# **Интеграционное тестирование**

На этапе *интеграционного тестирования* отдельные программные модули объединяются и тестируются в группе. Обычно интеграционное тестирование проводится после модульного тестирования. Основной целью его проведения является поиск багов, связанных с ошибками в реализации и интерпретации взаимодействия между модулями. Основная разница между модульным и интеграционным тестированием состоит в типах обнаруживаемых дефектов. В частности, на уровне интеграционного тестирования часто применяются методы, связанные с покрытием интерфейсов. Интеграционное тестирование использует метод “белого ящика” на модульном уровне.

Выделяют два уровня интеграционного тестирования:

* Компонентный интеграционный уровень (*Component Integration testing*). В этом уровне проверяется взаимодействие между компонентами системы после проведения модульного тестирования.
* Системный интеграционный уровень (*System Integration testing*). Здесь проверяется взаимодействие между разными системами после проведения системного тестирования.

Как правило, интеграционное тестирование проводится уже по завершении модульного тестирования для всех интегрируемых модулей. Однако это далеко не всегда так. Существует несколько методов проведения интеграционного тестирования:

* Восходящее тестирование
* Монолитное тестирование
* Нисходящее тестирование

Все эти методики основываются на знаниях об архитектуре системы, которая часто изображается в виде структурных диаграмм или диаграмм вызовов функций. Каждый узел на такой диаграмме представляет собой программный модуль, а стрелки между ними представляют собой зависимость по вызовам между модулями. Основное различие методов интеграционного тестирования заключается в направлении движения по этим диаграммам и в широте охвата за одну итерацию. Рассмотрим подробнее каждый из методов:

## **Восходящее тестирование:**

При использовании этого метода сначала тестируются все программные модули, входящие в состав системы, а затем они объединяются для проведения интеграционного тестирования. При использовании такого подхода значительно упрощается поиск ошибок, так как если модули протестированы по отдельности, то ошибка при их совместной работе есть проблема их интерфейса. При таком подходе область поиска проблем у тестировщика достаточно узка, поэтому вероятность правильно идентифицировать дефект гораздо выше. Однако у восходящего метода есть и недостатки, главный из которых состоит в необходимости разработки драйвера и заглушек для модульного тестирования перед проведением интеграционного тестирования и необходимость в разработке драйвера и заглушек при интеграционном тестировании части модулей системы. С одной стороны драйверы и заглушки – мощный инструмент тестирования, с другой – их разработка требует значительных ресурсов, особенно при изменении состава интегрируемых модулей, иначе говоря, может потребоваться один набор драйверов для модульного тестирования каждого модуля, отдельный драйвер и заглушки для тестирования интеграции двух модулей из набора, отдельный – для тестирования интеграции трех модулей и т.п. В первую очередь это связано с тем, что при интеграции модулей отпадает необходимость в некоторых заглушках, а также требуется изменение драйвера, которое поддерживает новые тесты, затрагивающие несколько модулей.

## **Монолитное тестирование:**

При использовании монолитного тестирования предполагается, что отдельные компоненты системы серьезного тестирования не проходили. Основное преимущество данного метода состоит в отсутствии необходимости в разработке тестового окружения, драйверов и заглушек. После разработки всех модулей выполняется их интеграция, затем система проверяется вся в целом. На первый взгляд кажется, что монолитное тестирование имеет много общего с системным тестированием. Однако несмотря на то, что при монолитном тестировании проверяется работа всей системы в целом, основная задача этого тестирования – определить проблемы взаимодействия отдельных модулей системы. Задачей же системного тестирования является оценка качественных и количественных характеристик системы с точки зрения их приемлемости для конечного пользователя.

Однако несмотря на все преимущества, монолитное тестирование имеет и ряд серьезных недостатков:

* Часто очень трудно выявить источник ошибки, так как одновременно тестируется несколько модулей и в каждом из них следует предполагать наличие ошибки. В итоге проблема сводится к определению того, какая из ошибок во всех вовлеченных модулях привела к полученному результату. При этом возможно наложение эффектов ошибок, а также ошибка в одном модуле может блокировать тестирование другого.
* Трудно организовать исправление ошибки, поскольку, как правило, тестируемые модули написаны разными людьми. Из-за этого возникает проблема – кто является ответственным за поиск и устранения этой ошибки? В следствие такой проблемы скорость устранения ошибки может значительно снизиться.
* Процесс тестирования плохо автоматизируется. В итоге преимущество в виде отсутствия дополнительного программного обеспечения оборачивается недостатком, так как каждое внесенное изменение требует повторения всех тестов.

## **Нисходящее тестирование:**

При использовании нисходящего тестирования предполагается, что процесс интеграционного тестирования движется следом за обработкой. Сначала тестируют только самый верхний управляющий уровень системы, без модулей более низкого уровня. Затем постепенно с более высокоуровневыми модулями интегрируются более низкоуровневые. В результате применения такого метода отпадает необходимость в драйверах (роль драйвера выполняет более высокоуровневый модуль системы), однако сохраняется нужда в заглушках.

У разных специалистов в области тестирования разные мнения по поводу того, какой из методов более удобен при реальном тестировании программных систем. Йордан доказывает, что нисходящее тестирование наиболее приемлемо в реальных ситуациях, а Майерс полагает, что каждый из подходов имеет свои достоинства и недостатки, но в целом восходящий метод лучше. Вне зависимости от применяемого метода интеграционного тестирования, необходимо учитывать степень покрытия интеграционными тестами функциональности системы.

# **Особенности тестирования для ООП**

Несмотря на то, что тестирование является достаточно независимым процессом, применимым к программному обеспечению, разработанному с помощью любого метода проектирования, объектно-ориентированный подход все же привносит свои особенности:

* Естественной единицей тестирования является класс. Разбиение его на более мелкие элементы (методы) нецелесообразно, поскольку они не существуют отдельно от классов. Иногда за единицу тестирования принимается тесно связанная группа классов.
* Тестирование наследования состоит в тестировании методов, унаследованных классом от своего базового класса. Если базовый класс уже прошел тестирование, необходимо повторить тестирование для унаследованных методов, так как они выполняются в новом контексте.
* При тестировании полиморфизма необходимо в тестовых сценариях предусмотреть все варианты связывания, то есть все варианты конкретной реализации полиморфизма.
* В данном случае метод интеграционного тестирования основан на выделении кластеров классов, имеющих вместе некоторую замкнутую и законченную функциональность. По своей сути такой подход не является новым типом интеграционного тестирования, просто меняется минимальный элемент, получаемый в результате интеграции. При интеграции модулей на процедурных языках программирования можно интегрировать любое количество модулей при условии разработки заглушек. При интеграции классов в кластеры существует достаточно нестрогое ограничение на законченность функциональности кластера. Однако, даже в случае объектно-ориентированных систем возможно интегрировать любое количество классов при помощи классов-заглушек.

# **Особенности индустриального тестирования**

Индустриальное тестирование подразумевает более тщательный подход к процессу тестирования и разбиение его на следующие фазы:

1. *Определение целей* (требований к тестированию). На этой фазе решается, какие части системы будут исследоваться, какие аспекты их работы будут выбраны для проверки, каково желаемое качество и т.п.
2. *Планирование*: создание графика разработки тестов для каждой тестируемое подсистемы; оценка необходимых человеческих, программных и аппаратных ресурсов; разработка расписания тестовых циклов.
3. *Разработка тестов*, то есть тестового кода для тестируемой системы и, если необходимо – кода системы автоматизации тестирования и тестовых процедур, выполняемых вручную.
4. *Выполнение тестов*: реализация тестовых циклов.
5. *Анализ результатов*.

После анализа результатов возможно повторение процесса тестирования, начиная с пунктов 3, 2 или даже 1.

## **Тестовый цикл**

*Тестовый цикл* – это цикл исполнения тестов, включающий фазы 4 и 5 тестового процесса. Тестовый цикл заключается в прогоне разработанных тестов на некотором однозначно определяемом срезе системы (состоянии кода разрабатываемой системы). Обычно такой срез системы называют *build*. Тестовый цикл включает следующую последовательность действий:

1. *Проверка готовности системы и тестов* к проведению тестового цикла.
2. *Подготовка тестовой машины* в соответствии с требованиями.
3. *Воспроизведение среза системы.*
4. *Прогон тестов.*
5. *Сохранение тестовых протоколов (test log)*.
6. *Анализ протоколов тестирования* и принятие решения о том, прошел или не прошел каждый из тестов (*Pass / Fail*).
7. *Анализ и документирование* результатов цикла.

## **Тестовый план**

*Тестовый план* – это документ, или набор документов, содержащий следующую информацию:

1. Тестовые ресурсы.
2. Перечень функций и подсистем, подлежащих тестированию.
3. Тестовую стратегию, включающую:
   1. Анализ функций и подсистем с целью определения наиболее слабых мест.
   2. Определение стратегии выбора входных данных для тестирования.
   3. Определение потребности в автоматизированной системе тестирования и дизайн такой системы.
4. Расписание тестовых циклов.
5. Фиксацию тестовой конфигурации: состава и конкретных параметров аппаратуры и программного окружения.
6. Определение списка тестовых метрик, которые на тестовом цикле необходимо собрать и проанализировать.

Также неотъемлемыми частями индустриального тестирования являются *составление документации*, *сопровождение тестов* и *оценка качества тестов*. Тесты нуждаются в контроле качества так же, как и тестируемый продукт. Поскольку тесты для продукта являются своего рода эталоном его структурных и поведенческих характеристик, закономерен вопрос о том, насколько адекватен эталон. Для оценки качества тестов используются различные методы, наиболее популярными из которых являются *тестовые метрики* и *обзоры тестов*.

# **Регрессионное тестирование**

*Регрессионное тестирование* – собирательное название для всех видов тестирования программного обеспечения, направленных на обнаружение ошибок в уже протестированных участках исходного кода. Такие ошибки, когда после внесения изменений в программу перестаёт работать то, что должно было продолжать работать, называются *регрессионными ошибками* (*regression bugs*).

Регрессионное тестирование включает *new bug-fix* — проверка исправления вновь найденного дефекта, *old bug-fix* — проверка, что исправленный ранее и верифицированный дефект не воспроизводится в системе снова, а также *side-effect* — проверка того, что не нарушилась работоспособность работающей ранее функциональности, если её код мог быть затронут при исправлении некоторых дефектов в другой функциональности. Обычно используемые методы регрессионного тестирования включают повторные прогоны предыдущих тестов, а также проверки, не попали ли регрессионные ошибки в очередную версию в результате слияния кода.

Из опыта разработки ПО известно, что повторное появление одних и тех же ошибок — случай достаточно частый. Иногда это происходит из-за слабой техники управления версиями или по причине человеческой ошибки при работе с системой управления версиями. Но настолько же часто решение проблемы бывает «недолго живущим»: после следующего изменения в программе решение перестаёт работать. И наконец, при переписывании какой-либо части кода часто всплывают те же ошибки, что были в предыдущей реализации.

Поэтому считается хорошей практикой при исправлении ошибки создать тест на неё и регулярно прогонять его при последующих изменениях программы. Хотя регрессионное тестирование может быть выполнено и вручную, но чаще всего это делается с помощью специализированных программ, позволяющих выполнять все регрессионные тесты автоматически. В некоторых проектах даже используются инструменты для автоматического прогона регрессионных тестов через заданный интервал времени. Обычно это выполняется после каждой удачной компиляции (в небольших проектах) либо каждую ночь или каждую неделю.

Регрессионное тестирование может быть использовано не только для проверки корректности программы, часто оно также используется для оценки качества полученного результата. Так, при разработке компилятора при прогоне регрессионных тестов рассматривается размер получаемого кода, скорость его выполнения и время компиляции каждого из тестовых примеров.

Существует следующая классификация регрессионного тестирования:

* **Тест минимизации наборов** (*test suite minimization*) стремится уменьшить размер тестового набора за счет устранения избыточных тестовых примеров из тестового набора.
* **Тестовая задача на определение приоритетов** (*test case prioritization*). Ее цели заключаются в выполнении заказанных тестов на основе какого-либо критерия. Например, на основе истории, базы или требований, которые, как ожидается, приведут к более раннему выявлению неисправностей или помогут максимизировать некоторые другие полезные свойства.
* **Тестовая задача выбора** (*test case selection*) связана с проблемой выбора подмножества тестов, которые будут использоваться для проверки измененных частей программного обеспечения. Для этого требуется выбрать подмножество тестов из предыдущей версии, которые могут обнаруживать неисправности, основываясь на различных стратегиях. Большинство задокументированных методов регрессионного тестирования сосредоточены именно на этой технике.
* **Гибридный тест**. Является сочетанием задач на определение приоритетов и выбора.

## **Преимущества и недостатки регрессионного тестирования**

Регрессионное тестирование выполняется при внесении изменений в существующие функциональные возможности программного обеспечения или, если есть ошибка исправления в программном обеспечении. Регрессионное тестирование может быть реализовано за счет нескольких подходов. Прохождение модифицированной программой всех тестов успешно обеспечивает уверенность в том, что изменения, внесенные в программное обеспечение, не повлияли на существующие функциональные возможности, которые должны быть неизменными в любом случае.

В гибком процессе управления проектами, где жизненный цикл разработки программного обеспечения очень короткий, не хватает ресурсов, и изменения в программное обеспечение вносятся очень часто. Регрессионное тестирование может ввести много ненужных накладных расходов.

Как правило, регрессивное тестирование осуществляется с помощью средств автоматизации, но нынешнее поколение инструментов регрессионного тестирования не предназначено для обработки приложений баз данных. По этой причине при выполнении регрессионного теста на приложениях, использующих базы данных, могут возникнуть незапланированные траты, поскольку это потребует много ручного труда.

# **Список литературы**

1. Г. Майерс. Искусство тестирования программ. - М.: Финансы и статистика, 1982.

2. Тестирование объектно-ориентированного программного обеспечения, К: Диасофт, 2002. - 432с.

3. С. Макконнелл. Совершенный код. — СПб: «Питер», 2005. — 896 с.

4. Тестирование программ. М.: Радио и связь, 1986. - 296с.

5. Тестирование программного обеспечения. К: ДиаСофт, 2000 - 544с.

6. С. Канер, Д. Фолк, Е. Нгуен. Тестирование программного обеспечения. — К.: Диасофт, 2000. — 544 с.

# **Ресурсы**

1. [www.intuit.ru/catalog/se/testing](http://www.intuit.ru/catalog/se/testing) - курсы лекций

2. [www.intuit.ru/department/se/testing/](http://www.intuit.ru/department/se/testing/)

3. <http://www.protesting.ru/>

4. [ru.wikipedia.org](https://ru.wikipedia.org/) – Тестирование ПО, ISO 9126

|  |
| --- |
|  |