Министерство образования Республики Беларусь

Учреждение образования
«Гомельский государственный технический
университет имени П. О. Сухого»

Институт повышения квалификации
и переподготовки кадров

Кафедра «Информатика»

ТЕСТИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

КУРС ЛЕКЦИЙ
по одноименной дисциплине
для слушателей специальности 1-40 01 73
«Программное обеспечение
информационных систем»
заочной формы обучения

Гомель 2014

УДК 004.43(075.8)
ББК 32.973-018я73
Т36

Рекомендовано научно-методическим советом
факультета автоматизированных и информационных систем
ГГТУ им. П. О. Сухого
(*протокол № 5 от 30.12.2013 г.*)

Составитель: Т. Л. Романькова

Рецензент: доц. каф. «Информационные технологии» ГГТУ им. П. О. Сухого
канд. физ.-мат. наук *О. А. Кравченко*

Тестирование программного обеспечения : курс лекций по одноим. дисциплине для
Т36 слушателей специальности 1-40 01 73 «Программное обеспечение информационных систем» заоч. формы обучения / сост. Т. Л. Романькова. – Гомель : ГГТУ им. П. О Сухого, 2014. – 56 с. Систем. требования: PC не ниже Intel Celeron 300 МГц ; 32 Mb RAM ; свободное место на HDD 16 Mb ; Windows 98 и выше ; Adobe Acrobat Reader. – Режим доступа: http://library.gstu.by. – Загл. с титул. экрана.

Даны основные понятия и принципы тестирования программного обеспечения. Рассматриваются основные виды тестирования программ. Особое внимание уделяется модульному и интеграционному тестированию программных продуктов.

Изучение дисциплины проводится на базе объектно-ориентированного языка C#.

Для слушателей специальности 1-40 01 73 «Программное обеспечение информационных систем» заочной формы обучения ИПК и ПК.

УДК 004.43(075.8)
ББК 32.973-018я73

Введение

Основные задачи курса лекций по дисциплине «Тестирование программного обеспечения»:

–познакомить слушателей с основными понятиями тестирования;

–рассмотреть разновидности тестирования;

–познакомить слушателей с методами и приемами тестирования программных приложений, а также с критериями выбора тестов;

– дать представление об особенностях тестирования объектно-ориентированных программ.

Тестирование программного обеспечения является очень важным и трудоемким видом деятельности. Тестирование – это один из наиболее применяемых способов обеспечения качества разработки программного обеспечения и, с другой стороны, представляет собой одну из фаз процесса разработки программного продукта.

В данной работе будут рассмотрены основные виды тестирования программ и те способы тестирования, которые применяются непосредственно в процессе разработки программных продуктов, что как раз является актуальным для будущих программистов.

Изучение дисциплины проводится на базе объектно-ориентированного языка C#.

1. Основные понятия тестирования программного обеспечения.

1.1. Тестирование программного обеспечения и процесс обеспечения качества

Обеспечение качества (*Quality Assurance*) — совокупность планируемых и систематически осуществляемых процессов, процедур, операций и отдельных мероприятий, необходимых для создания уверенности в том, что продукция удовлетворяет определенным требованиям к качеству

Тестирование программного обеспечения (*Testing*) — процесс, помогающий определить корректность, полноту и качество разработанного продукта.

Тестирование представляет собой процесс выявления наличия дефектов в программных системах.

Тестирование позволяет добиться того, чтобы готовый программный продукт соответствовал требованиям технического задания, однако тестирование — это еще не обеспечение качества. Хотя некоторые программисты ошибочно отождествляют понятия тестирования и обеспечения качества.

Обеспечение качества — деятельность, ориентированная на *предотвращение появления дефектов*, равно как и устранение дефектов, которые вкрались в программный продукт.

Тестирование содействует тому, чтобы конкретное программное приложение делало именно то, что ожидают от него проектировщики, а в некоторых случаях, чтобы проектируемое приложение не делало ничего более того, для чего оно предназначено.

Общее в обеспечении качества и тестировании заключается в том, что *они призваны улучшить программное обеспечение*.

Различие между ними — в том, что:

- обеспечение качества призвано улучшить программное обеспечение через улучшение процесса его разработки;
- тестирование — через обнаружение дефектов.

Тестирование содействует распознаванию отказов, благодаря чему разработчики могут найти и устранить ошибки в программах. Таким образом, тестирование способствует повышению качества программного продукта, помогая выявлять проблемы на ранних стадиях процесса разработки.

Цель тестирования — это нахождение ***дефектов*** до того, как их найдут пользователи.

Но стопроцентная проверка и нахождение всех дефектов для реального программного обеспечения – из области мифологии. Кроме того, количество найденных до передачи продукта пользователю дефектов не может служить критерием эффективности тестирования.

Существует популярное убеждение, что программу можно полностью протестировать.

• В некоторых учебниках для начинающих даже рассказывается, как это сделать: «Проверьте реакцию программы на все возможные варианты входных данных или все возможные последовательности выполнения ее кода». Как правило, выполнить эти рекомендации вообще невозможно.

• В возможность проведения полного тестирования верят и многие менеджеры. Они требуют этого от своего персонала и по окончании работ уверяют друг друга, что задача выполнена.

• Коммерческие представители некоторых компаний, специализирующихся на тестировании программного обеспечения, обещают, что код заказчика будет полностью протестирован.

• При продаже некоторых систем автоматизации тестирования вам обещают, что система сообщит, когда код будет протестирован полностью, а в ходе работы будет подсказывать, какие еще тесты позволят выявить все оставшиеся ошибки.

• Многие продавцы программного обеспечения верят, что их товар полностью протестирован и не имеет ошибок, и горячо убеждают в этом покупателей.

• В миф о полном тестировании верят и некоторые несчастные тестировщики. Их гнетет вечное чувство вины, поскольку как бы тяжело они ни трудились, как бы тщательно ни планировали работу, сколько бы ни тратили времени, сколько бы сотрудников и техники ни было задействовано в тестировании, все равно в программах остаются ошибки.

Ниже приводятся причины, по которым полное тестирование не может быть выполнено никогда.

• Количество всех возможных комбинаций входных данных слишком велико, чтобы его можно было проверить полностью.

• Количество всех возможных последовательностей выполнения кода программы также слишком велико, чтобы его можно было проверить

полностью.

• Пользовательский интерфейс программы (включающий все возможные комбинации действий пользователя и его перемещений по программе) обычно слишком сложен для полного тестирования.

Хороший тестировщик не тот, кто выявит больше всего ошибок, и не тот, кто заставит смутиться даже самого первоклассного программиста. Лучшим является тот, кто добьется исправления наибольшего количества ошибок.

Подведя итог, можно сказать, что только тестированием невозможно обеспечить качество программного обеспечения. Обеспечение качества — это задача всех участников процесса разработки программного продукта.

1.2. Понятие дефекта

Любое тестирование — это поиск дефектов. Дефект может быть привнесен на стадии разработки или сопровождения, в результате чего появляется одна или большее число ошибок («багов»).

Баг (англ. *Bug* — жук, мелкое насекомое) — распространенное среди программистов название ошибок в программах.

Существуют различные версии возникновения этого сленгового термина.

Широко распространена легенда, что 9 сентября 1945 года учёные Гарвардского университета, тестировавшие вычислительную машину Mark II Aiken Relay Calculator, нашли мотылька, застрявшего между контактами электромеханического реле, и Грейс Хоппер произнесла этот термин. Извлечённое насекомое было вклеено скотчем в технический дневник, с сопроводительной надписью: «First actual case of bug being found» («первый реальный случай, когда был найден *жук*»). В действительности этот случай произошёл 9 сентября 1947, а не 1945, года.

Слово «bug» в современном значении употреблялось задолго до этого персоналом телеграфных и телефонных компаний в отношении неполадок с электрооборудованием и радиотехникой. Во время Второй мировой войны словом «bugs» назывались проблемы с радарной электроникой.

Дефект (bug) — это отклонение **фактического** результата от **ожидаемого** результата.

Как утверждает Роман Савин в своей книге «Тестирование Дот Ком, или Пособие по жестокому обращению с багами в интернет-стартапах», основными источниками ожидаемого результата являются:

1. Спецификация.
2. Спецификация.
3. Спецификация.
4. Спецификация.
5. Жизненный опыт, здравый смысл, общение, устоявшиеся стандарты, статистические данные, авторитетное мнение и др.

Спецификация (**spec**) — это детальное описание того, как должно работать программное обеспечение.

В большинстве случаев дефект — это отклонение от спецификации.

По точке приложения дефекты можно разделить на:

- Ошибки пользовательского интерфейса.
- Ошибки функциональности.
- Ошибки логики программирования.
- Ошибки инсталляции.
- Ошибки использования памяти, системных ресурсов и т.д.

1.3. Место тестирования в жизненном цикле проекта и цикл тестирования программного продукта

В жизненном цикле проекта можно выделить следующие стадии:

- Осознание потребности в информационной системе.
- Формирование требований.
- Проектирование системы.
- Кодирование.
- Тестирование.
- Эксплуатация и поддержка.

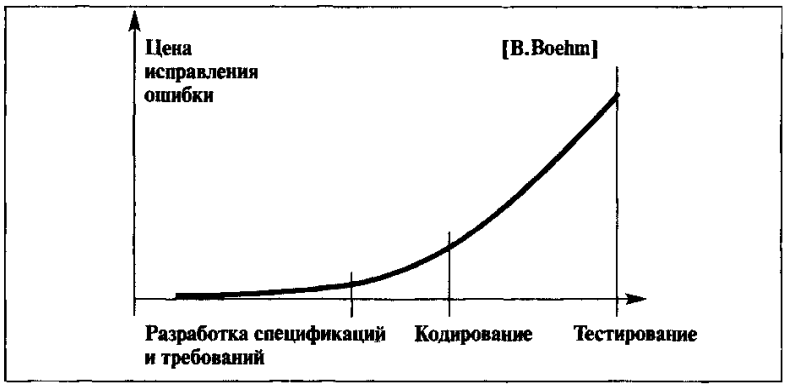


Рис.1.1. Оценка трудоемкости обнаружения и исправления ошибок при создании программного продукта

Тестирование является одной из основных фаз процесса разработки программного продукта и характеризуется большим вкладом в суммарную трудоемкость разработки. На рис. 1.1 отображена широко известная оценка распределения трудоемкости между фазами создания программы: 40% - 20% - 40%.

Наибольший эффект в снижении трудоемкости может быть достигнут на стадиях кодирования и тестирования. На рис.1.2. приведена диаграмма вариантов использования, которая показывает, на каких этапах создания программного продукта включаются в работу специалисты по тестированию.

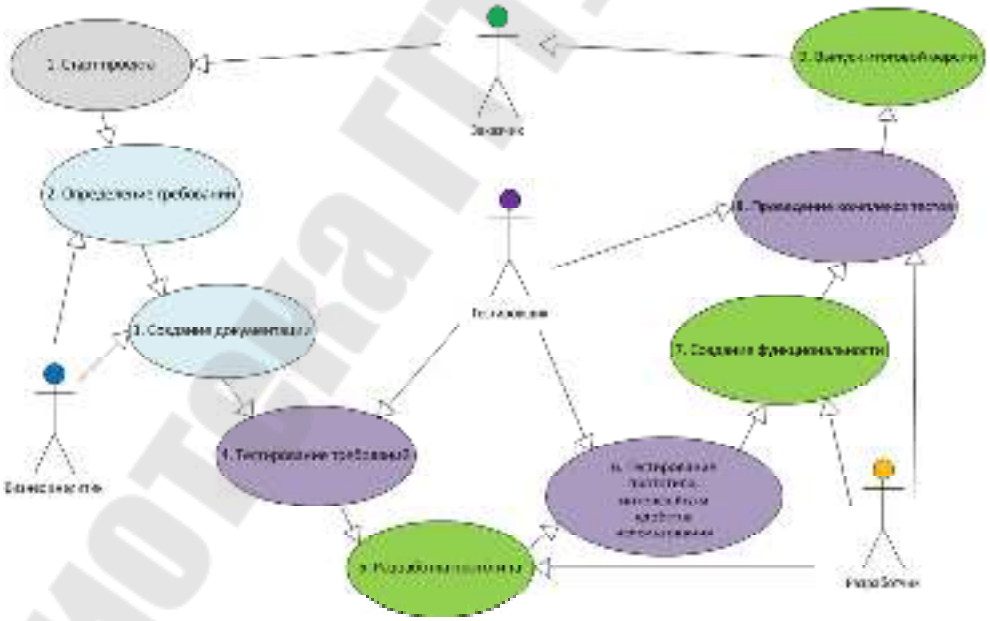


Рис.1.2. Жизненный цикл IT проекта

Цикл тестирования программного обеспечения состоит из следующих этапов (рис.1.3):

- Изучение и анализ предмета тестирования.
- Планирование тестирования.
- Разработка тестов.
- Выполнение тестирования.
- Анализ и отчет о результатах тестирования.

Рис.1.3. Цикл тестирования ПО

Изучение и анализ предмета тестирования начинаются перед утверждением спецификации и продолжаются на стадии «Кодирование». Планирование тестирования происходит на стадии «Кодирование». Исполнение тестирования происходит на стадии «Тестирование».

1.4. Ответственность команды тестирования

Тестированием на проекте могут и должны заниматься все участники проекта. Дело в том, что представители различных ролей на проекте протестируют продукт совершенно с разных точек зрения – менеджер и клиент найдут определенную категорию ошибок, дизайнеры найдут другую категорию ошибок, разработчики, аналитики и представители с иными задачами способны найти другие категории ошибок. Но в большинстве случаев нужна отдельная роль тестировщика. Ведь тестировщик со своим критическим мышлением способен докопаться до самых «глубоких» дефектов. Разделение ответственности помогает сфокусировать усилия на тестировании, а также получить независимую информацию от профессионала в тестировании.

Специалист по тестированию определяет стратегию тестирования, тест-требования и тест-планы для каждой из фаз проекта, выполняет тестирование системы, собирает и анализирует отчеты о прохождении тестирования. Тест-требования должны покрывать системные требования, функциональные спецификации, требования к надежности и нагрузочной способности, пользовательские интерфейсы и собственно программный код. В реальности роль специалиста по тестированию часто разбивается на две – разработчика тестов и тестиров-

щика. Тестировщик выполняет все работы по выполнению тестов и сбору информации, разработчик тестов – всю остальные работы.

В зону ответственности команды тестирования проекта входит:

- планирование времени и объемов тестирования;
- контроль бюджета проекта и сроков выполнения заданий;
- контроль качества проекта на всех стадиях разработки;
- взаимодействие с заказчиком и командой на стороне заказчика;
- создание тестовой документации;
- проведение теста;
- внесение дефектов;
- оформление тестовой отчетности.

Обязанности команды тестирования:

- понимать желания конечного пользователя;
- докапываться до сути проблем;
- оценивать работу любого приложения недоверчиво;
- описывать проблему так, чтобы ее поняли другие участники команды;
- быть в курсе новейших технологий и стремиться узнать больше для повышения своей квалификации.

1.5. Тестовая документация и отчетность

Поскольку тестирование программной системы выполняется в течение всего жизненного цикла разработки достаточно большим коллективом разработчиков, при тестировании создается тестовая документация. Основное ее назначение, помимо синхронизации действий тестировщиков различных уровней – обеспечение гарантий того, что тестирование выполняется в соответствии с выбранными критериями оценки качества, а также того, что все аспекты поведения системы протестированы. Кроме того, тестовая документация используется при внесении изменений в систему для проверки того, что как старая, так и новая функциональность работает корректно.

Не существует **обязательных** для исполнения правил о том, как эффективно протестировать ПО, какие документы нужно создать и в какой форме они должны быть. В различных источниках по-разному описывается процесс документирования тестирования. В разных компаниях существуют разные виды и формы документов.

Ниже приводится описание и примеры некоторых документов.

План тестирования – организационный документ, содержащий требования к тому, как должно выполняться тестирование в данном конкретном проекте. В данном документе также определяются требования собственно к тестовой документации – тест-требованиям, тест-планам, отчетам о выполнении тестирования и др..

Тест-требования – документы, в которых подробно описано то, какие аспекты поведения системы должны быть протестированы.

Как правило, структура тест-требований следует структуре раздела функциональных требований на систему. Задача каждого требования - определение того, **что** надо проверить. Техника исполнения каждой такой проверки - задача тест-плана. Обычный формат описания отдельного требования следующий:

Проверить, что при <описание внешнего воздействия> [происходит] <описание реакции программы >.

ст-планы – документы, которые содержат подробное пошаговое описание того, как должны быть п**Те** ротестированы тест-требования. Подробный план тестирования состоит из **тестовых сценариев**. Один тестовый случай, например, это проверка корректности работы одной кнопки, одного логически завершенного фрагмента функционала и т.д. На рис.1.4 приведен пример набора тестовых сценариев, разработанного в компании **Itransition.**

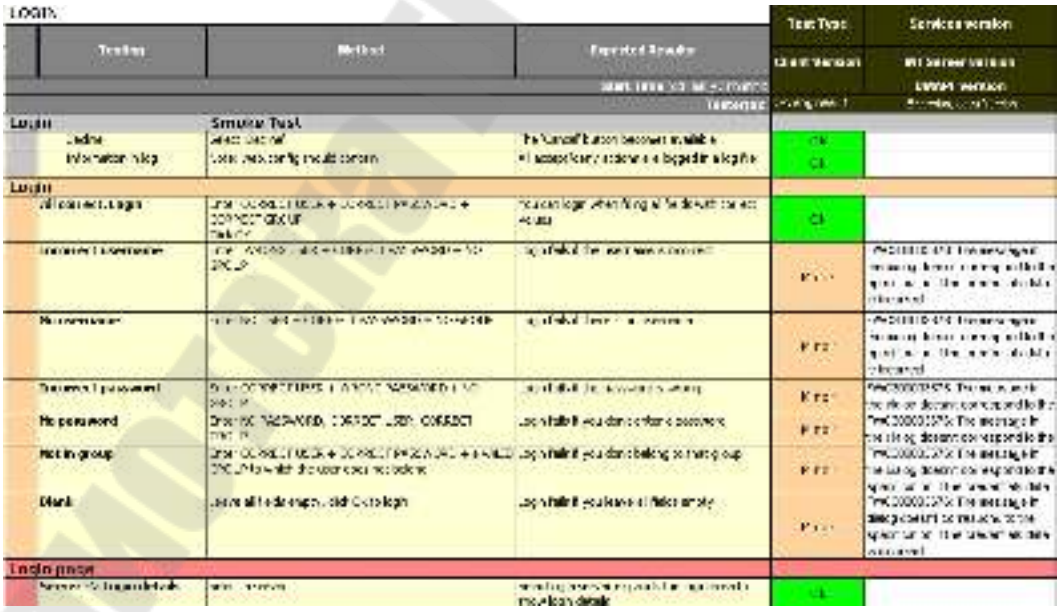


Рис.1.4. Набор тестовых сценариев

Тестовый случай (*test case*) или **тест-кейс** может иметь структуру, показанную на рисунке 1.5 из книги Романа Савина.

TC ID/Priority	CCPG0001	1
IDEA: Оплата может быть произведена картой VISA SETUP and ADDITIONAL INFO: Эккаунт: testuserl/paSSwOrd Наименование товара: bookl 17 Данные карты: Номер: 9999-5148-2222-1277 Окончание действия: 12/07 CVV2: 778 SQL1: select result from ccjxansaction where id = <номер заказа>;		
Revision History		
Created on: 11/17/2003 by O.Тарасов	Новый тест-кейс	
Execution part		
PROCEDURE		**EXPECTED RESULT**
1. Открой www.main.testshop.rs 2. Введи имя пользователя. 3. Введи пароль. 4. Нажми кнопку "Войти". 5. Введи наименование товара в поле поиска. 6. Нажми кнопку "Найти". 7. Кликни линк "Добавить в корзину". 8. Кликни линк "Корзина". 9. Кликни линк "Оплатить". 10. Выбери вид карты. 11. Введи номер карты. 12. Введи срок окончания действия. 13.ВведиСУУ2. 14. Нажми кнопку "Завершить заказ". 15. Запиши номер заказа 16. Запроси базу данных с SQL1 и запиши результат		> "10"

Рис.1.5. Пример тестового случая

Описание дефекта может иметь следующую структуру:

1. Headline: суть проблемы.
2. Severity: степень серьёзности дефекта.
3. Description: алгоритм воспроизведения дефекта.
4. Expected result: ожидаемое поведение.
5. Attachment: вспомогательное средство передачи информации о проблеме.

Например,

***Headline**: Полный путь к файлу не отображается на форме выбора приложения*

Description:

Шаги воспроизведения:

1. *Запустите систему*
2. *Откройте вкладку «Пациенты»*

3. *Выберите добавление/редактирование любого пациента из списка*
4. *Выберите действие «Добавить приложение»*
5. *Выберите любой файл*

Результат: отображается только имя файла

Expected result: *Имя файла и путь должны отображаться в соответствии с требованием ID1*

По результатам выполнения тестов создаются **отчеты о выполнении тестирования** (они могут создаваться либо автоматически, либо вручную), которые содержат информацию о том, какие несоответствия требованиям были выявлены в результате тестирования, а также отчеты о покрытии, содержащие информацию о том, какая доля программного кода системы была задействована в результате выполнения тестирования.

Отчет о качестве - отчетное письмо о проделанной работе и качестве проекта.

Основные параметры хорошего **отчета**:

- Информация о проведенных тестах и/или их специфике
- Информация о качестве проекта и его основных модулей
- Информация о прогрессе/регрессе качества с указанием областей и возможных причин
- Информация о качестве разработки новых функций и исправлении дефектов
- Аргументированность (чаще всего в виде статистики)
- Прозрачность преподнесенной информации для людей, не вовлеченных напрямую в процесс разработки и тестирования.

На рис.1.5 приведен пример отчета о качестве, составленного в компании **Itransition.**

Изменения в систему вносятся только после всестороннего изучения этих отчетов и локализации проблем, вызвавших несоответствие требованиям. Для того, чтобы процесс изменений не вышел из под контроля и любое изменение протоколировалось (и связывалось с тестами, обнаружившими проблему), создается запрос на изменение системы. После завершения всех работ по запросу на изменение процесс тестирования повторяется до тех пор, пока не будет достигнут приемлемый уровень качества программной системы.

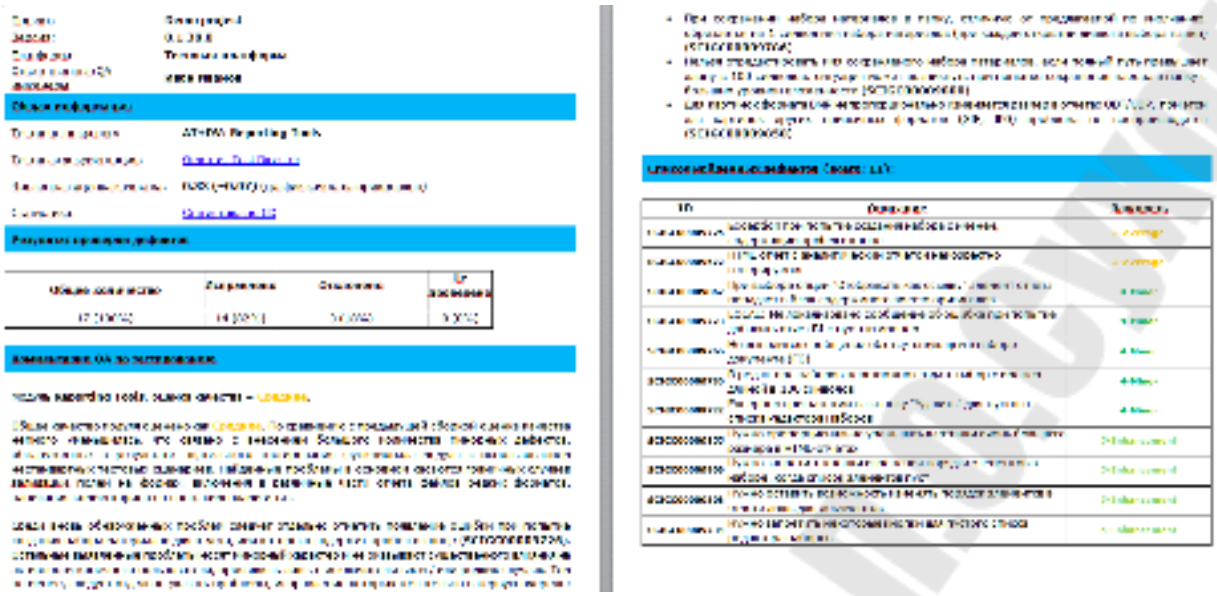


Рис.1.5. Пример отчета о качестве

2. Виды и уровни тестирования

2.1. Классификация видов тестирования

Выделяют следующие признаки, по которым классифицируют виды тестирования:

- По знанию системы:
  - черный ящик (*black box testing*);
  - серый ящик (*grey box testing*);
  - белый ящик (*white box testing*).
- По объекту тестирования:
  - тестирование требований;
  - функциональное тестирование (*functional testing*);
  - тестирование интерфейса пользователя (*UI testing*);
  - тестирование локализации (*localization testing*);
  - тестирование скорости и надежности (*load/stress/performance testing*);
  - тестирование безопасности (*security testing*);
  - тестирование опыта пользователя или удобства использования (*usability testing*);
  - тестирование совместимости (*compatibility testing*);
  - тестирование программ установки и лицензирования;
  - тестирование доступности.

- По времени проведения тестирования:
  - до передачи пользователю — альфа-тестирование (*alpha-testing*);
  - после передачи пользователю — бета-тестирование (*beta testing*).
- По признаку «позитивности» сценариев:
  - позитивное тестирование (*positive testing*);
  - негативное тестирование (*negative testing*).
- По степени изолированности тестируемых компонентов:
  - компонентное тестирование (*component testing*);
  - интеграционное тестирование (*integration testing*);
  - системное тестирование (*system or end-to-end testing*).
- По степени автоматизации тестирования:
  - ручное тестирование (*manual testing*);
  - автоматизированное тестирование (*automated testing*);
  - смешанное/полуавтоматизированное тестирование.
- По степени подготовки к тестированию:
  - тестирование по документации (*formal/documented testing*);
  - эд хок-тестирование (*ad hoc testing*).

2.2. Классификация по знанию системы

По знанию внутренностей системы можно выделить следующие виды тестирования:

- тестирование «Белого ящика»;
- тестирование «Черного ящика»;
- тестирование «Серого ящика».

Основная идея в тестировании системы, как черного ящика состоит в том, что все материалы, которые доступны тестировщику – требования на систему, описывающие ее поведение и сама система, работать с которой он может только подавая на ее входы некоторые внешние воздействия и наблюдая на выходах некоторый результат. Все внутренние особенности реализации системы скрыты от тестировщика, таким образом, система и представляет собой «черный ящик», правильность поведения которого по отношению к требованиям и предстоит проверить.

При подходе **«Черный ящик»** тестировщик не основывает идеи для тестирования на знании об устройстве и логике тестируемой час-

ти системы. Идеи формируются путем предположений о сценариях, которые будут реализовываться и применяться пользователями.

Такие сценарии называются *паттернами поведения* пользователей.

Подход «Черный ящик» также называют поведенческим.

При подходе «**Белый ящик**» (стеклянный, открытый, никакой) тестировщик основывает идеи для тестирования на знании об устройстве и логике тестируемой части системы.

При тестировании системы, как стеклянного ящика, тестировщик имеет доступ не только к требованиям на систему, ее входам и выходам, но и к ее внутренней структуре – видит ее программный код. Доступность программного кода расширяет возможности тестировщика тем, что он может видеть соответствие требований участкам программного кода и видеть тем самым – на весь ли программный код существуют требования. Программный код, для которого отсутствуют требования называют кодом, непокрытым требованиями. Такой код является потенциальным источником неадекватного поведения системы. Кроме того, прозрачность системы позволяет углубить анализ ее участков, вызывающих проблемы – часто одна проблема нейтрализует другую и они никогда не возникают одновременно.

При «белоящичном» тестировании сценарии создаются для того чтобы протестировать определенную часть кода, а не определенный паттерн поведения пользователя.

Вот как иллюстрирует описанные подходы к тестированию Р.Савин в книге «Тестирование Дот Ком, или Пособие по жестокому обращению с багами в интернет-стартапах».

Допустим, нужно протестировать проходимость нового российского внедорожника.

При подходе "Черный ящик" тестировщик садится за руль, выезжает за кольцевую — в объятия подмосковной осени, находит непролазную канаву, заезжает в нее и пытается выбраться, т.е. он проделывает вещи, которые с большой вероятностью будут проделаны основными пользователями таких машин — охотниками, рыболовами и т.д.

При подходе "Белый ящик" тестировщик открывает капот и видит, что установлена система полного привода фирмы "Джапан моторз", модель RT6511. Тестировщик знает, что проходимость внедорожника зависит именно от RT6511 и ее слабое место — это эффективность при езде по снегу. Что делает тестировщик? Пра-

вильно! Выезжает на белую сверкающую гладь русского поля и проверяет джип в определенных условиях.

Подход «Серый ящик» сочетает элементы двух предыдущих подходов. Это:

• с одной стороны, *тестирование, ориентированное на пользователя*, а значит, мы используем паттерны поведения пользователя, т.е. применяем методику «Черного ящика»;

• с другой — *информированное тестирование*, т.е. мы знаем, как устроена хотя бы часть тестируемого кода, и активно используем это знание.

2.3. Классификация по объекту тестирования

По объекту тестирования выделяют следующие виды тестирования:

- тестирование требований;
- функциональное тестирование;
- тестирование удобства использования;
- тестирование интерфейса пользователя;
- тестирование производительности;
- тестирование безопасности;
- тестирование локализации;
- тестирование совместимости;
- тестирование программ установки и лицензирования;
- тестирование доступности.

Тестирование требований - выявление несоответствий в ожиданиях и интерпретациях создаваемого продукта на ранних этапах разработки.

Ошибки в спецификациях могут перенестись в код и тесты, дефект, найденный раньше, стоит компании дешевле, и поэтому учет таких дефектов является важной обязанностью специалистов по тестированию.

Преимущества, получаемые в результате тестирования требований:

- Сокращение сроков сдачи готового продукта.
- Повышение качества продукта.
- Снижение затрат на разработку и тестирование продукта.

Тестирование интерфейса пользователя - это тестирование, при котором проверяются элементы интерфейса пользователя.

Тестирование локализации – это проверка множества аспектов, связанных с адаптацией программного обеспечения для пользователей из других стран. Например, тестирование локализации для пользователей из Японии может заключаться в проверке возможности ввода иероглифов.

Тестирование производительности – это проверка поведения приложения (или его отдельных частей) при одновременной работе множества пользователей.

Скорость и надежность программного продукта профессионально проверяется специальным программным обеспечением, например, Silk Performer от Segue или Load Runner от Mercury Interactive.

Тестирование удобства использования призвано объективно оценить опыт пользователя (user experience), который будет работать с разрабатываемым интерфейсом. Такое тестирование часто проводится путем привлечения группы потенциальных пользователей с целью собрать впечатления от работы с системой.

Тестирование безопасности — это стратегия тестирования, используемая для проверки безопасности системы, а также для анализа рисков, связанных с обеспечением целостного подхода к защите приложения от атак хакеров, вирусов, несанкционированного доступа к данным.

Тестирование совместимости – это проверка того, как тестируемый программный продукт взаимодействует с техническим и программным обеспечением пользователя.

Функциональное тестирование – это проверка работы функциональностей.

Функциональность (*functionality, feature*) — это средство для решения некой задачи.

Важность функционального тестирования состоит в том, что функциональности — это не что иное, как продукт, предоставляемый пользователям.

Функциональное тестирование основывается на функциях, выполняемых системой. Эти функции описываются в требованиях, функциональных спецификациях или в виде случаев использования системы (*use case*). Преимуществом этого вида тестирования является то, что оно имитирует фактическое использование системы. В то же время к недостаткам функционального тестирования можно отнести

возможность упущения логических ошибок в программном обеспечении и вероятность избыточного тестирования.

2.4. Классификация по степени автоматизации

По степени автоматизации тестирование делится на:

- ручное;
- автоматизированное;
- полуавтоматизированное.

Ручное тестирование - это тестирование без помощи каких-либо программ, автоматизирующих работу специалиста.

Автоматизированное тестирование программного обеспечения (*Software Automation Testing*) - это процесс проверки программного обеспечения, при котором основные функции и шаги теста, такие как запуск, инициализация, выполнение, анализ и выдача результата, выполняются автоматически при помощи инструментов для автоматизированного тестирования.

Существует два основных подхода к автоматизации тестирования:

- тестирование на уровне кода (например, модульное тестирование);
- GUI-тестирование (имитация действий пользователя).

В таблице 1 приводятся некоторые инструменты для автоматизированного функционального тестирования.

Таблица 1.

Компания	**Инструмент**
Hewlett-Packard (Mercury Interactive)	QuickTest Professional, WinRunner
IBM Rational	Rational Robot, Rational Functional Tester
Borland (Segue)	SilkTest
AutomatedQA Corp	TestComplete
Microsoft	Microsoft VS 2005
OpenQA	Selenium

Автоматические тесты не могут полностью заменить ручное тестирование. Автоматизация всех испытаний — очень дорогой про-

цесс, и потому автоматическое тестирование является лишь дополнением ручного тестирования.

Полуавтоматизированное – это тестирование, в котором ручной подход сочетается с автоматизированным.

2.5. Классификация по времени проведения тестирования

По времени проведения тестирования выделяют

- альфа-тестирование;
- бета-тестирование.

Альфа-тестированием называется любое тестирование кода **до** передачи его пользователям.

Бета-тестирование — интенсивное использование почти готовой версии продукта (как правило, программного или аппаратного обеспечения) с целью выявления максимального числа ошибок в его работе для их последующего устранения перед окончательным выходом (релизом) продукта на рынок, к массовому потребителю.

Идея **бета-тестирования** заключается в следующем: перед открытием нового кода всем пользователям, новый код открывается лишь ограниченной группе пользователей, которые его и протестируют.

Эти пользователи («бета-тестировщики») должны являться представителями целевой аудитории, для удовлетворения потребностей которой и был разработан код.

Бета-тестировщики, являясь типичными пользователями, будут делать с бета-версией приложения те же вещи, что и остальные пользователи после официального представления, и, следовательно, заранее столкнутся с «непойманными багами», о которых они и сообщат разработчикам.

Бета-тестирование может использоваться как часть стратегии продвижения продукта на рынок, а также для получения предварительных отзывов о нём от широкого круга будущих пользователей.

2.6. Классификация по признаку позитивности сценариев

По признаку позитивности сценариев различают

- позитивное тестирование;
- негативное тестирование.

Сценарий, проверяющий ситуацию, связанную с потенциальной ошибкой пользователя и/или потенциальным дефектом в системе, называется **негативным**.

Создание и исполнение тестов с негативными сценариями называется негативным тестированием. При этом виде тестирования проверяется, как ведет себя приложение, получая на вход «неправильные» данные, и показывается то, что программное обеспечение работает неправильно (и при каких случаях).

Основной целью «негативного» тестирования является проверка устойчивости системы к воздействиям различного рода, валидация неверного набора данных, проверка обработки исключительных ситуаций.

Позитивные сценарии — это сценарии, предполагающие нормальное, «правильное» использование и/или работу системы.

Создание и исполнение тестов с позитивными сценариями называется позитивным тестированием.

В ходе проведения данного вида тестирования программного обеспечения проверяется результат работы приложения при получении им «правильных» входных данных и показывается, что программное обеспечение работает правильно.

Основной целью «позитивного» тестирования является проверка того, что при помощи системы можно делать то, для чего она создавалась.

«Позитивное» тестирование считается более важным, чем «негативное», т.к. ошибки в нормальном поведении, мешающие пользователям выполнять бизнес задачи обходятся в разы дороже, чем редкие падения системы, связанные с тем, что кто-то ввел некорректные данные.

Пользователи рано или поздно научатся обходить «подводные камни», не будут делать «опасные» или «неразрешённые» действия, служба технической поддержки скоро запомнит, какие проблемы обычно возникают у пользователей и будет давать советы типа «ни в коем случае не оставляйте это поле пустым».

2.7. Классификация по степени подготовленности к тестированию

По данному признаку выделяют следующие виды тестирования:

- тестирование по документации;
- интуитивное тестирование.

Тестирование приложения по документации (*formal testing*) проводится по заранее подготовленным данным (тест-кейсы, чек-листы, чит-листы, спецификация и т.д.).

Интуитивное (*ad hoc*, с лат. — для этой цели), тестирование, как правило, применятся в качестве теста приемки и/или теста сдачи (если тестовые случаи для них не формализованы в документации); в качестве дополнения к документированным тестированию новых функциональностей и регрессивному тестированию; в других случаях, когда нет описания тестов.

Преимущества формализованного (scripted) тестирования:

• обеспечивает разделение работ;

• интеграция различных техник тестирования;

• тест кейсы можно однозначно отлинковать к требованиям;

• тесты детализированы, они легче поддаются автоматизации;

• тесты создаются на ранних этапах в процессе разработке;

• тест кейсы по окончанию проекта становятся де-факто спецификацией требований;

• обнаружение неисследованных ранее областей.

Недостатки:

• напрямую зависит от качества требований;

• по своему определению, не гибкое. Оно всегда следует согласно запланированному перечню действий;

• часто используется для снижения роли тестировщика.

2.8. Классификация по степени изолированности компонентов

По степени изолированности компонентов различают:

- модульное (компонентное) тестирование;
- интеграционное;
- системное.

Компонентное тестирование — это тестирование программы на уровне отдельно взятых модулей, функций и классов.

Цель модульного тестирования — изолировать отдельные части программы и показать, что по отдельности эти части работоспособны.

Этот тип тестирования обычно выполняется *программистами*.

Обычно компонентное (модульное) тестирование проводится вызывая код, который необходимо проверить при поддержке сред разработки, таких как фреймворки (*frameworks* - каркасы) для модульного тестирования или инструменты для отладки.

Модульное тестирование проводится по принципу белого ящика.

Все найденные **дефекты**, как правило исправляются в коде без формального их описания в системе менеджмента багов (Bug Tracking System) .

Интеграционное тестирование – это тестирование части системы, состоящей из двух и более модулей.

Системное тестирование — это тестирование программного обеспечения, выполняемое на полной, интегрированной системе, с целью проверки соответствия системы исходным требованиям.

Системное тестирование проводится над проектом в целом с помощью метода черного ящика.

На данном этапе тестировщика интересует не корректность реализации отдельных процедур и методов, а вся программа в целом, как ее видит конечный пользователь.

Можно выделить два подхода к системному тестированию:

- на базе требований (requirements based): для каждого требования пишутся тестовые случаи (test cases), проверяющие выполнение данного требования;
- на базе случаев использования (use case based).

Существуют следующие категории системного тестирования:

- Тестирование полноты решения функциональных задач.
- Стрессовое тестирование – на предельных объемах нагрузки входного потока.
- Проверка корректности использования ресурсов.
- Оценка производительности.
- Тестирование эффективности защиты от искажения данных и некорректных действий.
- Проверка инсталляции и конфигурации на разных платформах.
- Корректность документации.

2.9. Глубина тестирования сборки

До передачи приложения пользователю в различное время проводят следующие виды проверки программного продукта:

- Приемочный тест;
- Полный тест;
- Регрессионный тест;

- Проверка исправленных дефектов;
- Тестирование новых функций.

Приемочный тест - поверхностный тест, проверяющий основную функциональность программного продукта и его работоспособность, по результатам которого руководитель команды тестирования принимает решение о приемке версии программного продукта на дальнейшее тестирование.

Smoke Test (дымовое тестирование) означает минимальный набор тестов на явные ошибки. Дымовой тест обычно выполняется самим программистом; не проходящую этот тест программу не имеет смысла отдавать на более глубокое тестирование.

Первое свое применение этот термин получил у печников, которые, собрав печь, закрывали все заглушки, затапливали её и смотрели, чтобы дым шёл только из положенных мест.

Полный тест включает в себя:

- Тестирование всех функций и модулей, в том числе и проверку на некорректных значениях, а также проверку нестандартного использования программного продукта;
- Проверку всех взаимосвязей и логических завязок;
- Проверку общей логики проекта и отсутствие противоречий в работе элементов системы;
- Тестирование на соответствие документации;
- Тестирование интерфейса пользователя;
- Тестирование удобства использования.

Тестирование новых функций включает:

- Проведение полного теста непосредственно по новой функциональности;
- Проверку общей логики проекта, всевозможных завязок ранее реализованной функциональности с новыми модулями и функциями.

Проверка исправленных дефектов - это проверка на воспроизводимость дефектов, которые были исправлены в новой сборке продукта.

Регрессионное тестирование – это проверка стабильности работы функций, которые работали ранее.

Регрессионное тестирование предполагает повторное проведение тестов, в результате которых была обнаружена ошибка, после исправления дефекта, чтобы убедиться, что ошибок больше нет. В дан-

ном случае задача регрессионного тестирования состоит в том, чтобы убедиться, что выявленная ошибка полностью исправлена программистом и больше не проявляется. Каждый раз, когда в программу вносится изменение, все тесты проводятся снова. Особенно важно провести такое обстоятельное тестирование, если программа изменяется спустя достаточно длительное время или новым программистом.

Вот еще один пример применения регрессионного тестирования. После выявления и исправления ошибки проводится стандартная серия тестов, но уже с другой целью: убедиться, что исправляя одну часть программы, программист не испортил другую. В этом случае тестируется целостность всей программы, а не исправление одной ошибки.

3. Модульное тестирование

3.1. Задачи и цели модульного тестирования

Для того чтобы удостовериться в корректной работе системы в целом, необходимо вначале протестировать каждый модуль системы по отдельности. Такое тестирование модулей по отдельности получило название **модульного или компонентного тестирования** (*unit testing*).

Цель *модульного тестирования* - выявить локализованные в модуле ошибки в реализации алгоритмов, удостовериться в соответствии требованиям каждого отдельного модуля системы перед тем, как будет произведена его интеграция в состав системы.

Основные задачи, решаемые в процессе компонентного тестирования:

1. Поиск и документирование несоответствий требованиям. Сюда входит разработка тестового окружения и тестовых примеров, выполнение тестов, протоколирование результатов выполнения, составление отчетов о проблемах.
2. Поддержка разработки и рефакторинга низкоуровневой архитектуры системы и межмодульного взаимодействия.
3. Поддержка рефакторинга модулей. Модульные тесты при этом являются мощным инструментом для проверки того, что новый вариант программного кода выполняет те же функции, что и старый.

4. Поддержка устранения дефектов и отладки. Подробные отчеты о проблемах, составленные на этапе модульного тестирования, позволяют локализовать и устранить многие дефекты в программной системе на ранних стадиях ее разработки или разработки ее новой функциональности.

Рефакторинг — это процесс улучшения написанного ранее кода путем такого изменения его внутренней структуры, которое не влияет на внешнее поведение.

3.2. Основные понятия

Модульное тестирование - это тестирование программы на уровне отдельно взятых модулей.

Модульный тест представляет собой программу, занимающуюся тестированием некоего модуля.

Модуль, в контексте модульного тестирования, представляет собой минимальную смысловую единицу исходного кода, пригодную для тестирования (функция, процедура, метод, в отдельных случаях – класс).

Обычно за тестируемый модуль принимается либо программный модуль (единица компиляции) в случае, если система разрабатывается на процедурном языке программирования или класс, если система разрабатывается на объектно-ориентированном языке.

В случае систем, написанных на процедурных языках, процесс тестирования модуля происходит так– для каждого модуля разрабатывается тестовый драйвер, вызывающий функции модуля и собирающий результаты их работы и набор заглушек, которые имитируют поведение функций, содержащихся в других модулях, не попадающих под тестирование данного модуля.

В случае объектно-ориентированных систем более мелкое деление классов и использование отдельных методов в качестве тестируемых модулей нецелесообразно в связи с тем, что для тестирования каждого метода потребуется разработка тестового окружения, сравнимого по сложности с уже написанным программным кодом класса.

Модульное тестирование является основой для всех других видов тестирования. Модульный тест проверяет работоспособность модуля в условиях изоляции от других модулей приложения.

На уровне *модульного тестирования* проще всего обнаружить дефекты, связанные с алгоритмическими ошибками и ошибками ко-

дирования алгоритмов (работа с условиями и счетчиками циклов), а также с использованием локальных переменных и ресурсов.

Ошибки, связанные с неверной трактовкой данных, некорректной реализацией интерфейсов, совместимостью, производительностью и т.п. обычно пропускаются на уровне *модульного тестирования* и выявляются на более поздних стадиях тестирования.

Один из наиболее эффективных подходов к компонентному (модульному) тестированию - это **подготовка автоматизированных тестов** до начала основного кодирования (разработки) программного обеспечения. Это называется разработка от тестирования (*test-driven development - TDD*) или подход тестирования вначале (*test first approach*).

При этом подходе создаются и интегрируются небольшие куски кода, для которых запускаются тесты, написанные до начала кодирования. Разработка ведется до тех пор пока все тесты не будут успешными. Модульные тесты – это инструмент, а TDD – методология, один из вариантов их использования.

Аргументы в пользу модульных тестов:

1. Лучше качество кода. *По сути, комбинация «основного» кода и модульных тестов – это двойная запись одной и той же функциональности. В коде могут быть ошибки. В тесте могут быть ошибки. Но вероятность ошибки в коде и в тесте гораздо меньше.*

2. Возможность проводить рефакторинг без опасений поломать работу приложения и погрязнуть в отладке. *К примеру, маленькое «улучшение» в каком-то алгоритме не разрушит всю систему, если на алгоритм и смежные компоненты есть модульные тесты.*

3. Скорость нахождения дефектов.

По оценкам экспертов, модульные тесты позволяют найти около 15% ошибок, выявленных в ходе полного цикла разработки. В то же время, большей частью это критические баги, и их раннее выявление способно сберечь существенную долю ресурсов и предотвратить катаклизмы в среде пользователя программного продукта.

4. Возможность тестирования базовой функциональности без интерфейса пользователя.

5. Более удобная координация работы распределенной группы программистов. *Модульные тесты позволяют провести приемочные испытания перед интегрированием модулей в единую систему.*

6. Модульные тесты служат дополнением к документации и помогают новому разработчику войти в курс дела.

7. Точечная настройка производительности системы и обнаружение утечек памяти. *Воспроизвести 100 тыс. одновременных подключений и выполнение определенной последовательности действий в системе не так-то просто. Гораздо удобнее воспроизвести эту нагрузку с помощью модульных тестов.*

Аргументы против модульных тестов:

1. Время – деньги. *Вместо написания кода, который можно продать, разработчик будет трудиться над тестами. Иногда вместо того, чтобы проверять все подряд, лучше по факту посмотреть, что сломалось, и чинить именно эту часть.*

2. Модульных тестов недостаточно для качественного тестирования приложений. *Модульные тесты не охватывают работу всей системы в целом. Они проверяют систему с позиций разработчика и не могут взглянуть на приложение глазами пользователя.*

3. Модульные тесты выполняются в стерильных условиях. *Заглушки и имитаторы – удобная штука, но это лишь эмуляция поведения системы. В реальной жизни код может валиться, скажем, из-за проблем с производительностью, причем именно в том месте, где, казалось бы, модульные тесты и не нужны.*

4. Модульные тесты покажут наличие ошибок, но не докажут их отсутствие.

3.3. Принципы построения модульных тестов

Модульное тестирование является по способу исполнения структурным тестированием или тестированием «белого ящика».

Тесты, связанные со структурным тестированием, строятся по следующим принципам:

- На основе анализа *потока управления*.

В этом случае элементы, которые должны быть покрыты при прохождении тестов, - вершины, дуги, пути управляющего графа программы (УГП), условия, комбинации условий и т. п.

- На основе анализа *потока данных*, когда элементы, которые должны быть покрыты, определяются на основе *потока данных*, т. е. информационного графа программы.

Управляющий граф программы (УГП) отображает поток управления программы. Это граф G(V, A), где V(V1,… Vm) – множеств вершин (операторов), A(A1,… An) – множество дуг (управлений), соединяющих вершины.

Путь – последовательность вершин и дуг УГП, в которой любая дуга выходит из вершины Vi и приходит в вершину Vj.

Ветвь – *путь*(V1, V2, … Vk), где V1 - либо первый, либо условный оператор, Vk - либо условный оператор, либо оператор выхода из программы, а все остальные операторы – безусловные.

Число путей в программе может быть не ограничено (*пути, различающиеся хотя бы числом прохождений цикла – разные*). Ветви - линейные участки программы, их *конечное* число.

Существуют реализуемые и нереализуемые пути в программе, в нереализуемые пути в обычных условиях попасть нельзя.

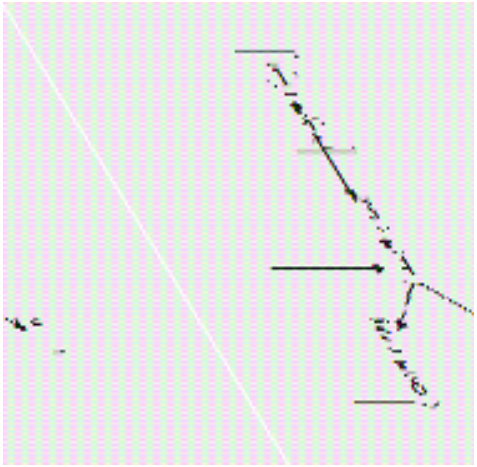
Например, в методе

```
float Func(float x, float y)
{
        float H;
        if (x*x+y*y+2<=0)
              H = 17;
        else       H = 64;
       return    H*H+x*x;
}
```

путь (1,2,4) нереализуем.

Пример.

```
/* Функция вычисляет неотрицательную
   степень n числа x */
1. Public double Power(double x, int n)
   {
2. double z= 1; int i;
3. for ( i = 1;     i <= n;     i++ )
4. { z = z* x; }  /* Возврат в п.4 */
5. return z;
   }
```



Узлы 1,2 не включаются в УГП, т.к. не содержат управляющих операторов.

Примеры путей: (3,4,7), (3,4,5,6,4,5,6), (3,4), (3,4,5,6).

Примеры ветвей: (3,4) (4,5,6,4) (4,7).

На этапе модульного тестирования используются **структурные критерии тестирования**:

Тестирование команд (критерий **C0**) - набор тестов в совокупности должен обеспечить прохождение каждой команды не менее одного раза.

Тестирование ветвей (критерий **C1**) - набор тестов в совокупности должен обеспечить прохождение каждой ветви не менее одного раза.

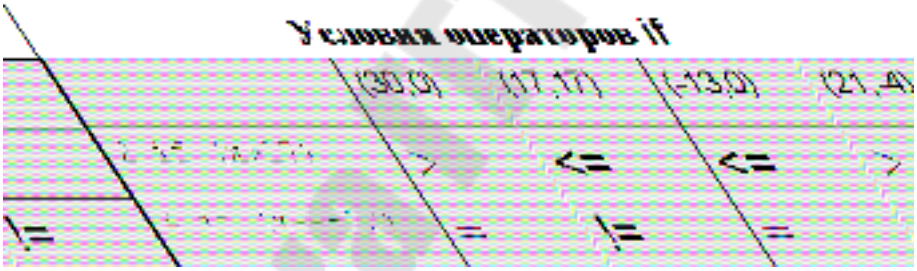
Тестирование путей (критерий **C2**) - набор тестов в совокупности должен обеспечить прохождение каждого пути не менее 1 раз. Если программа содержит цикл (в особенности с неявно заданным числом итераций), то число итераций ограничивается константой.

Тестирование условий - покрытие всех булевских условий в программе.

Критерии покрытия решений (ветвей - C1) и условий не заменяют друг друга, поэтому на практике используется комбинированный критерий покрытия условий/решений, совмещающий требования по покрытию и решений, и условий.

Пример

```
1  public void Method (ref int x) {
2  if (x>17)
3        x = 17-x;
4  if (x==-13)
5        x = 0;
6  }
```



По критерию команд (C0):

Тестовый набор (вх, вых) = {(30, 0)} удовлетворяет критерию C0 – покрываются все операторы трассы 1-2-3-4-5-6.

По критерию ветвей (C1): {(30, 0), (17, 17)}

Трасса 1-2-3-4-5-6 проходит через все ветви достижимые в операторах *if* при условии *true*, а трасса 1-2-4-6 через все ветви, достижимые в операторах *if* при условии *false*.

По критерию путей (C2):({(30,0), (17,17), (-13,0), (21,-4)}

Структурные критерии не проверяют соответствие спецификации**,** если оно не отражено в структуре программы.

Например, если спецификация задает условие, что **|x|<100**, невыполнимость этого условия можно подтвердить на тесте (-177,-177).

Процесс построения набора тестов при структурном тестировании принято делить на три фазы:

- **Конструирование УГП.**

Задача получить граф программы и зависящее от него и от критерия тестирования множество элементов, которые необходимо покрыть тестами.

- **Выбор тестовых путей**.

Выделяют три подхода к построению тестовых путей:

- Статические методы.
- Динамические методы.
- Методы реализуемых путей.

- Генерация тестов, соответствующих тестовым путям.

По известным путям тестирования осуществляется поиск подходящих тестов, реализующих прохождение этих путей.

Методы построения множества тестов:

Статические методы. Построение каждого пути посредством постепенного его удлинения за счет добавления дуг, пока не будет достигнута выходная вершина графа программы.

Основной недостаток *статических методов* заключается в том, что не учитывается возможная нереализуемость построенных путей тестирования.

Динамические методы. Построение полной системы тестов, удовлетворяющих заданному критерию, путем одновременного решения задачи построения покрывающего множества путей и тестовых данных.

Методы реализуемых путей. Выделение из множества путей подмножества всех реализуемых путей, из которых строится покрывающее множество путей.

3.4. Построение тестового драйвера

Тестирование классов обычно выполняется путем разработки **тестового драйвера**, который создает экземпляры классов и окружает эти экземпляры соответствующей средой, чтобы стал возможен прогон соответствующего тестового случая.

В обязанности тестового драйвера обычно входит удаление любого созданного им экземпляра, если в языке программирования имеет место управляемое программистом распределение памяти (например, C++).

Тестовый драйвер представляет собой элемент программы, который осуществляет прогон тестовых случаев и сбор полученных при этом результатов.

Способы реализации тестового драйвера:

- В виде метода тестируемого класса, который может быть вызван для выполнения и сбора результатов прогона каждого тестового случая.

Плюсы:

- Программный код драйвера находится в непосредственной близости к программному коду класса.

- Упрощается многократное использование кода драйвера (в силу наследования) для тестирования подклассов.

Минусы:

- Необходимость соблюдения осторожности при отделении программного кода драйвера от поставляемого программного обеспечения.

- В виде отдельного класса, в обязанности которого входит выполнение и сбор результатов для каждого тестового случая.

Для тестирования нужно создать экземпляры этого класса. Прогон тестовых случаев может осуществляться вызовом специально созданных для этого методов от имени объектов класса или непосредственно при создании экземпляров.

Плюсы:

- Легко осуществляется многократное использование драйвера при тестировании подклассов.

- Достигается максимально компактный рабочий код.

- Достигается максимальное быстродействие рабочего кода.

Минусы:

- Необходимость построения нового класса.

- Необходимость соблюдения осторожности при отображения изменений в классе на изменения в тестах.

Тестовый драйвер можно также реализовать в виде класса-наследника тестируемого класса. Такому тестовому драйверу будет доступна не только **public**, но и **protected** часть класса.

Тестовое окружение также может использоваться для отчуждения отдельных модулей системы от всей системы. Разделение модулей системы на ранних этапах тестирования позволяет более точно локализовать проблемы, возникающие в их программном коде. Для поддержки работы модуля в отрыве от системы тестовое окружение должно моделировать поведение всех модулей, к функциям или данным которых обращается тестируемый модуль.

Поскольку тестовое окружение само является программой (причем зачастую реализованной не на том языке программирования, на котором написана система), оно само должно быть протестировано. Целью тестирования тестового окружения является доказательство того, что тестовое окружение никаким образом не искажает выполнение тестируемого модуля и адекватно моделирует поведение системы.

Пример.

Составляем спецификацию на класс «**Студенты**».

Класс Студенты

```
class Студенты
    {   Студент[] students; //ссылка на массив студентов
        //конструктор
      public Студенты(DataGridView DGV)

        //метод, формирующий список отличников
        public string[] Otlichniki(string grupp)

    }
```

Класс реализует поток студентов**.**

Операции:

Конструктор **Студенты**(DataGridView DGV) принимает на вход объект класса DataGridView и записывает данные из таблицы в поле класса, представляющее собой массив объектов класса **Студент**.

Метод **Otlichniki(string grupp)** формирует список студентов заданной группы, средний балл которых больше 8. Если группа не задана (на входе пустая строка), в группе нет отличников или заданная группа отсутствует на потоке, в качестве результата должны выдаваться соответствующие сообщения.

Спецификация тестового случая для тестирования
метода Otlichniki:

Название класса: Студенты	Название тестового случая: СтудентыTest_1
Описание тестового случая: Тест проверяет правильность работы метода **Otlichniki** – формирования списка отличников заданной группы. Для следующего потока Иванов ИТ-21 3 Петров ИТ-21 9 Сидоров ИТ-22 10 Минаев ИТ-21 9,5 Смехов ИТ-23 4 Тамерлан ИТ-23 5,9 Мухин ИТ-22 5 Кривенко ИТ-23 6,7 Шестаков ИТ-21 6,3 На вход подаются следующие названия групп: **ИТ-21, ИТ-22, ТМ-13, ИТ-23, пустая строка**	
Ожидаемый результат: Группе ИТ-21 должен соответствовать список: **Петров, Минаев** Группе ИТ-22 должен соответствовать список: **Сидоров** Группе ТМ-13 должно соответствовать сообщение **Нет такой группы** Группе ИТ-23 должно соответствовать сообщение **В группе нет отличников** Пустой строке должно соответствовать сообщение **Не задана группа**	

Тестовый драйвер **Студенты_Tester**

```
class Còóäåí òû _Tester
   {  Ñòóäåí òû  rezult;
      StreamWriter log;
  public Còóäåí òû _Tester(string testfile_name)
       { StreamReader f = new StreamReader(testfile_name);
            string[] s;
         DataGridView dgv = new DataGridView();
 dgv.Columns.Add("c1",      "ô àì  èëèÿ");      dgv.Columns.Add("c2",
"ãðóï à");
  dgv.Columns.Add("c3", "ñðåäí èé áàëë");
        int i = 0;
              while (!f.EndOfStream)
```

```
{ s = f.ReadLine().Split();
   dgv.Rows.Add();
      dgv[0,i].Value = s[0];
      dgv[1,i].Value = s[1];
      dgv[2, i].Value = s[2];
      i++;
    }
f.Close();
    rezult = new Ñòóäåí òû (dgv);
   log = new StreamWriter("logFile.dat");
 Run( );
 }
void Run()
    { Ñòóäåí òû Test_1( ); log.Close( ); }
  void Ñòóäåí òû Test_1()
    { string[ ] gruppy ={"ÈÒ-21","ÈÒ-22","ÒÌ -13", "ÈÒ-23", "" };
      string[] s;
      for (int i = 0; i < 4; i++)
                {
                 log.WriteLine("òåñò ¹  " + i+" Ãðóï ï à"+gruppy[i]);
                 s=rezult.Otlichniki(gruppy[i]);
                 for (int j = 0; j < s.Length; j++) log.WriteLine("  " + s[j]);
                }
      }
      }
```

На форме нужно разместить кнопку ***Тест*** и создать обработчик события:

```
private void button3_Click(object sender, EventArgs e)
    {
      Còóääí òû _Tester stud = new Còóääí òû _Tester("test1.txt");
    }
```

Сформированный Log-файл:

тест №0 Группа ИТ-21

Петров

Минаев

тест №1 Группа ИТ-22

Сидоров

тест №2 Группа ТМ-13

тест №3 Группа ИТ-23

Получили несоответствие со спецификацией! Необходимо описать дефекты.

Желательно, чтобы интерфейсная часть тестирующего класса содержала набор методов для построения экземпляров тестируемого класса.

Конкретный класс **Tester** должен реализовать метод, соответствующий каждому конструктору, определенному в тестируемом классе. Методы тестовых случаев используют эти библиотечные методы для построения объектов вместо конструкторов тестируемого класса.

Рекомендации по созданию модульных тестов:

- Не надо тестировать каждый класс. Только самые значимые. Модульные тесты призваны проверять базовую функциональность и наиболее «ценный» код.
- Особое внимание логике приложения (*if, for, while* и т.п.). Если у продукта будет отсутствовать меню, это будет заметно сразу. Проверить все логические ходы-выходы гораздо сложнее.
- Модульные тесты должны выполняться быстро, иначе в них нет смысла.

- Должны требовать минимальных изменений в случае изменения «основного» кода.

4. Интеграционное тестирование

4.1. Задачи и цели интеграционного тестирования

Отдельные модули редко функционируют сами по себе, поэтому следующая задача после тестирования отдельных модулей – тестирование корректности взаимодействия нескольких модулей, объединенных в единое целое. Такое тестирование называют **интеграционным**.

Цель *интеграционного тестирования* - удостовериться в корректности совместной работы компонент системы, проверить связи между компонентами, а также взаимодействия с различными частями системы (операционной системой, оборудованием либо связи между различными системами).

Основные задачи, решаемые в процессе компонентного тестирования:

1. Проверка всех возможных видов взаимодействий между программными модулями и элементами, которые определяются в архитектуре системы.
2. Проверка корректности взаимодействия компонент системы. Результаты выполнения интеграционных тестов – один из основных источников информации для процесса улучшения и уточнения архитектуры системы, межмодульных и межкомпонентных интерфейсов.

Интеграционное тестирование, как правило, представляет собой итеративный процесс, при котором проверяется функциональной все более и более увеличивающейся в размерах совокупности модулей.

4.2. Основные понятия

Интеграционное тестирование - одна из фаз тестирования программного обеспечения, при котором отдельные программные модули объединяются и тестируются в группе.

Обычно интеграционное тестирование проводится после модульного тестирования и предшествует системному тестированию. Интеграционное тестирование – это тестирование «белого ящика ».

Уровни интеграционного тестирования:

- **Компонентный интеграционный уровень** (*Component Integration testing*)

Проверяется взаимодействие между компонентами системы после проведения компонентного тестирования.

- **Системный интеграционный уровень** (*System Integration Testing*)

Проверяется взаимодействие между разными системами после проведения системного тестирования.

Взаимодействие объектов представляет собой запрос одного объекта (*отправителя*) на выполнение другим объектом (*получателем*) одной из операций получателя и всех видов обработки, необходимых для завершения этого запроса.

Типы взаимодействий:

Тип 1. Общедоступная операция именует один или большее число классов как тип формального параметра. Сообщение устанавливает ассоциацию между получателем и параметром, которая позволяет получателю взаимодействовать с этим параметрическим объектом.

Например, в классе ***Dekanat*** метод

public string Nakazanie(Student st)

служит иллюстрацией такого взаимодействия**.**

Тип 2. Общедоступная операция присваивает имена одному или большему количеству классов как тип возвращаемого значения.

Например, метод класса ***Dekanat***

public Student Registracia(string dannye)

иллюстрирует отношения такого рода.

Тип 3. Метод одного класса создает экземпляр другого класса как часть своей реализации. Например,

```
public Ñòóäåí òû(DataGridView DGV)
    {
      int n = DGV.RowCount-1;
      students=new Ñòóäåí ò[n];
      string s = "";
      for (int i = 0; i < n; i++)
      { s = DGV[0, i].Value.ToString()+" " + DGV[1, i].Value.ToString()+" " +
DGV[2, i].Value.ToString();

        students[i] = new Ñòóäåí ò(s);
      }
```

Тип 4. Метод одного класса ссылается на глобальный экземпляр некоторого другого класса.

Принципы хорошего тона в проектировании рекомендуют минимальное использование глобальных объектов. Если реализация какого-либо класса ссылается на некоторый глобальный объект, рассматривайте его как неявный параметр в методах, которые на него ссылаются.

Например, в классе **Univer** описан класс **Fakultet**, в котором есть метод, ссылающийся на поле **Rector** класса **Univer**.

4.3. Классификация методов интеграционного тестирования

С точки зрения структуры можно выделить следующие подходы к интеграционному тестированию:

- восходящее тестирование;
- монолитное тестирование;
- нисходящее тестирование.

Все эти методики основываются на знаниях об архитектуре системы.

Рассмотрим **Тестирование снизу вверх** (*Bottom Up Integration*) или **Восходящее тестирование**.

При таком подходе все низкоуровневые модули собираются вместе и затем тестируются. После чего собирается следующий уровень модулей для проведения интеграционного тестирования.

Данный подход считается полезным, если все или практически все модули, разрабатываемого уровня, готовы. Он помогает определить уровень готовности приложения, т.к. значительно упрощается локализация ошибок: если модули протестированы по отдельности, то ошибка при их совместной работе есть проблема их интерфейса.

У восходящего метода тестирования есть существенный недостаток – необходимость в разработке 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.

На рис.4.1. показана схема разработки драйверов и заглушек при восходящем тестировании.

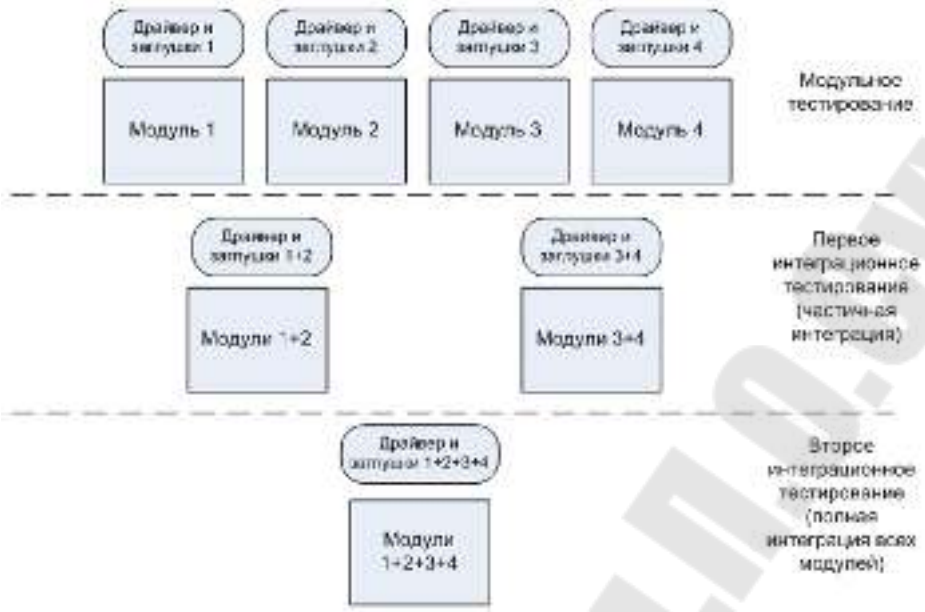


Рис.4.1. Разработка драйверов и заглушек при восходящем тестировании

Тестирование сверху вниз (*Top Down Integration*) или **нисходящее тестирование** предполагает, что вначале тестируются все высокоуровневые модули, и постепенно один за другим добавляются низкоуровневые.

Все модули более низкого уровня симулируются заглушками с аналогичной функциональностью, затем по мере готовности они заменяются реальными активными компонентами.

В результате применения такого метода отпадает необходимость в драйверах (роль драйвера выполняет более высокоуровневый модуль системы).

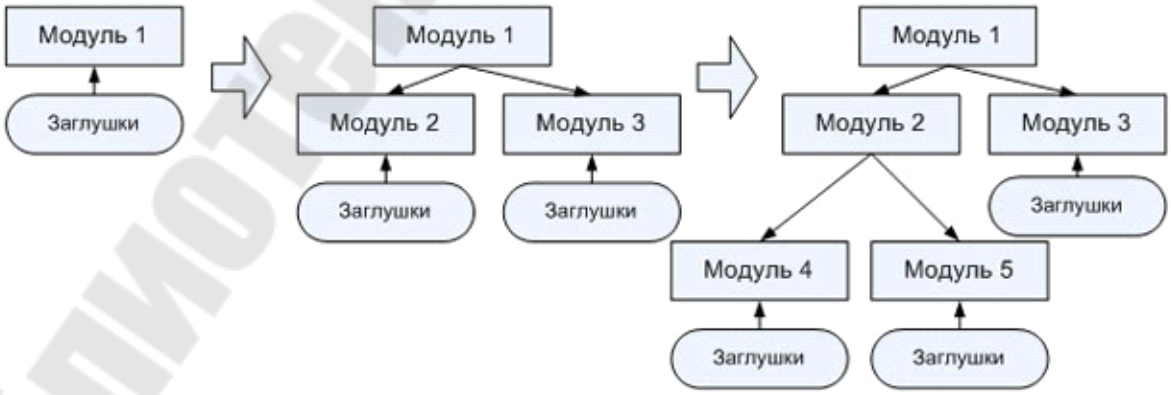


Рис.4.2. Постепенная интеграция модулей при нисходящем методе тестирования

Большой взрыв (*"Big Bang" Integration*) или **Монолитное тестирование** предполагает, что все или практически все разработанные модули собираются вместе в виде законченной системы или ее основной части, и затем проводится интеграционное тестирование.

Этот подход не следует путать с системным тестированием, которому посвящена следующая тема. Несмотря на то, что при монолитном тестировании проверятся работа всей системы в целом, основная задача этого тестирования – определить проблемы взаимодействия отдельных модулей системы.

Основное преимущество данного метода – отсутствие необходимости в разработке тестового окружения, драйверов и заглушек.

Монолитное тестирование имеет ряд серьезных недостатков:

- очень трудно выявить источник ошибки (идентифицировать ошибочный фрагмент кода);
- трудно организовать исправление ошибок;
- процесс тестирования плохо автоматизируется.

На практике чаще всего в различных частях проекта применяются все рассмотренные в предыдущем разделе методы в совокупности.

По частоте интеграции выделяют:

- тестирование с поздней интеграцией;
- тестирование с постоянной интеграцией;
- тестирование с регулярной или послойной интеграцией.

Тестирование с поздней интеграцией – практически полный аналог монолитного тестирования. Интеграционное тестирование при такой схеме откладывается на как можно более поздние сроки проекта.

Этот подход оправдывает себя в том случае, если система представляет собой совокупность слабо связанных между собой модулей. Схематично может быть изображено в виде цепочки

R-C-V-R-C-V-R-C-V-**I**-R-C-V-R-C-V-**I**,

где R – разработка требований на отдельный модуль,

C – разработка программного кода,

V – тестирование модуля,

I – интеграционное тестирование всего, что было сделано раньше.

Тестирование с постоянной интеграцией подразумевает, что как только разрабатывается новый модуль системы, он сразу же интегрируется со всей остальной системой.

Тесты для этого модуля проверяют как сугубо его внутреннюю функциональность, так и его взаимодействие с остальными модулями системы.

Разработки заглушек при таком подходе не требуется, но может потребоваться разработка драйверов.

Схематично тестирование с постоянной интеграцией может быть изображено в виде цепочки

R-C-**I**-R-C-I-R-C-**I**

Тестировании с регулярной или послойной интеграцией, при котором интеграционному тестированию подлежат сильно связанные между собой группы модулей (слои), которые затем также интегрируются между собой.

Такой вид интеграционного тестирования называют также иерархическим интеграционным тестированием. Направление прохода по иерархии в этом подходе не задано.

4.4. Возможности MVSTE по автоматизации модульного и интеграционного тестирования

В состав Microsoft Visual Studio Team System входит **Microsoft Visual Studio Team Edition** (MVSTE) for Software Testers.

MVSTE for Software Testers содержит следующие инструменты для тестирования:

- Manual Testing (ручное тестирование);
- Web Testing (функция веб-тестирования);
- Unit Testing (модульное тестирование);
- Code Coverage (анализ покрытия кода);
- Ordered Testing (упорядоченное тестирование);
- средства управления тестированием, позволяющие составлять тесты, выполнять их и централизованно отслеживать данный процесс.

Последовательность построения тестов в MVSTE:

- Щелкнуть правой кнопкой мыши на элементе класса, подлежащего тестированию.
- Выбрать пункт контекстного меню **Create Unit Tests...** (Создать модульные тесты…)

Появится диалоговое окно, позволяющее создать тесты в другом проекте, показанное на рисунке 4.3.

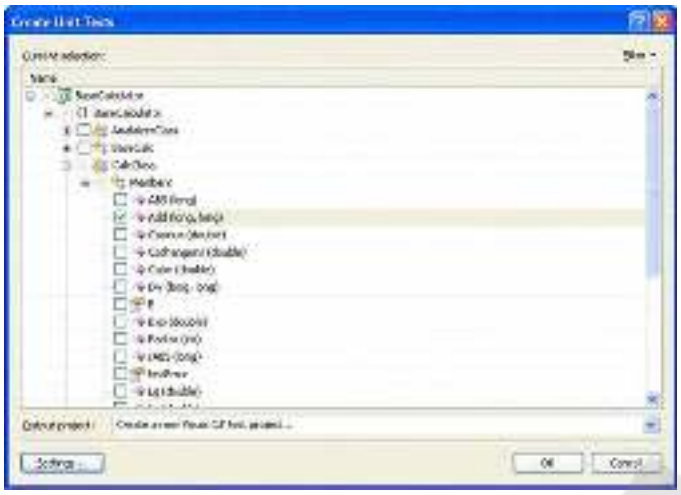


Рис.4.3. Окно **Create Unit Tests**

Сгенерированный код теста будет сильно зависеть от типа и сигнатуры того метода, который планируется тестировать.

В начале теста объявляется значение всех необходимых переменных, а также ожидаемое выходное значение.

Затем происходит вызов нужного метода, которому передаются необходимые параметры.

В конце размещается оператор
Assert.Inconclusive("Verify the correctness of this test method.");

Наличие этого метода в тесте говорит о том, что реализация теста еще не закончена.

- Внести изменения в тест и убрать обращение к методу Assert.Inconclusive("Verify the correctness of this test method.").
- Чтобы запустить тест, можно нажать кнопку на панели инструментов **Отладка тестов в текущем контексте.**

В окне **Test Results** будет показан результат выполнения теста.

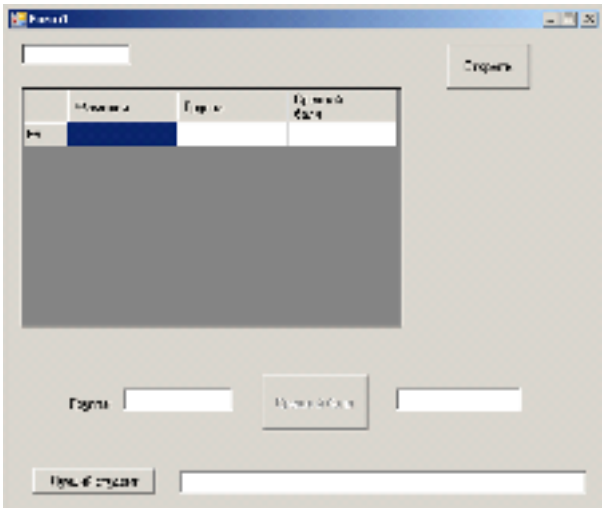
ПРИМЕР

Разработать Windows-приложение, которое выполняет следующие функции:

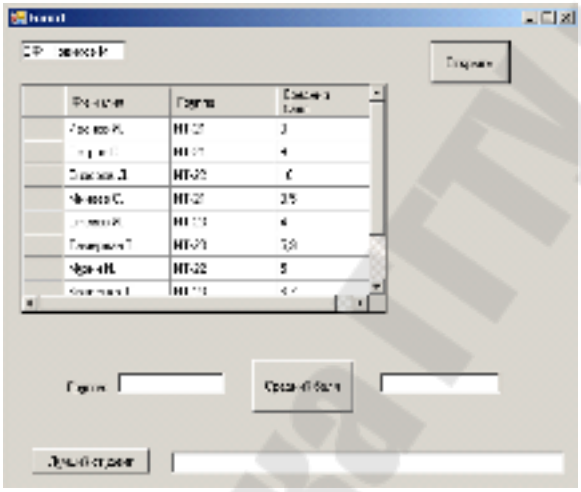
- Чтение из файла данных о студентах энергетического факультета (декан Новиков Михаил Иванович) при нажатии кнопки **Открыть** (использовать **стандартное** диалоговое окно Открыть); Строка файла содержит фамилию, имя, отчество и средний балл студента.
- вывод информации о студентах в таблицу;

- определение среднего балла заданной группы при нажатии соответствующей кнопки.
- Поиск лучшего студента при нажатии на кнопку

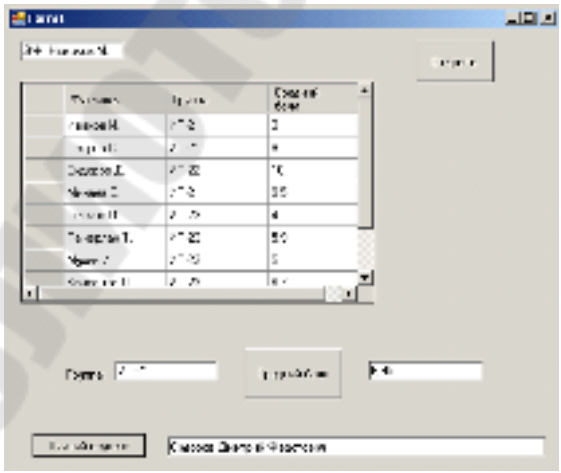
Окно приложения до нажатия кнопки «**Открть**»:



Окно приложения после нажатия кнопки «**Открыть**» и выбора файла:



Окно приложения после нажатия кнопок «**Средний балл**» и «**Лучший студент**»:



Информацию о факультете представить как объект класса **Fakultet**, содержащего поле-массив объектов класса **Student**. Фамилию, имя, отчество студентов и декана представлять как объект класса **Fio**.

Средствами **MVSTE** подготовить тестовый драйвер для класса **Fakultet.**

Класс FIO:

```
class Fio
   {
      string fam, name, otch;//ï î ëÿ
      public Fio(string fam, string name, string otch)//êî í ñòðóêòî ð
      {this.fam = fam; this.name = name; this.otch = otch;}
      public string Fam { get { return fam; } }//ñâî éñòâî  äëÿ äî ñòóï à ê ï î ëþ  fam
      public  string  Name  {  get  {  return  name;  }  }//ñâî éñòâî  äëÿ äî ñòóï à ê ï î ëþ  name
      public string Otch { get { return otch; } }//ñâî éñòâî  äëÿ äî ñòóï à ê ï î ëþ  Otch
   }
```

Описание класса Student:

```
class Student
   {
      Fio fio;
      string  gruppa;
      float s_ball;
public Student(Fio FIO,string gr,float sb)
      { fio=FIO;gruppa=gr;s_ball=sb;}
public Fio FIO { get { return fio; } }
      public string Fam { get { return fio.Fam; } }
      public string Name { get { return fio.Name; } }
      public string Gruppa { get { return gruppa; } }
      public float S_ball { get { return s_ball; }}       }
```

Описание класса Fakultet:

```
class Fakultet
  {  Student[] students;//ññû ëêà í à ì àññèâ ñòóäåí òî â
     Fio dekan;// äåêàí  ô àêóëüòåòà
     string Fname;//í àçâàí èå ô àêóëüòåòà
   public Fakultet(string fileName,string dekan,string fname)
        {
           StreamReader f = new StreamReader(fileName);
           string[] s=f.ReadToEnd().Split('\n');
           int n = s.Length;
           f.Close();
   students = new Student[n];
           f = new StreamReader(fileName);
           int i = 0;
           while (!f.EndOfStream)
           {
              s = f.ReadLine().Split();
              students[i] = new Student(new Fio(s[0], s[1], s[2]), s[3],
   Single.Parse(s[4]));
              i++;

           }
           f.Close();
this.dekan=new Fio(dekan.Split()[0],dekan.Split()[1],dekan.Split()[2]);
      Fname=fname;
      }
```

```
public float SBall_Gruppy(string grupp)
        {
           int k = 0; float S = 0,Sb;
        for (int i = 0; i < students.Length; i++)
        {
           if (students[i].Gruppa == grupp)
           { k++; S = S + students[i].S_ball; }
        }
       Sb=S/k;
            return Sb;
            }
```

и т.д.

Пример шаблона теста для проверки метода Sball_gruppy:

```
public void SBall_GruppyTest()
        {
            string fileName = string.Empty; // TODO: инициализация
подходящего значения
            string dekan = string.Empty; // TODO: инициализация подходящего
значения
            string fname = string.Empty; // TODO: инициализация подходящего
значения

            Fakultet target = new Fakultet(fileName, dekan, fname); // TODO:
инициализация подходящего значения
            string grupp = string.Empty; // TODO: инициализация подходящего
значения
            float expected = 0F; // TODO: инициализация подходящего значения
            float actual;
            actual = target.SBall_Gruppy(grupp);
            Assert.AreEqual(expected, actual);
            Assert.Inconclusive("Проверьте правильность этого метода
теста.");
        }
```

После внесения изменений в шаблон тест может выглядеть следующим образом:

```
public void SBall_GruppyTest()
        {
```

```
        string fileName = "D:\\test1.txt";
        string dekan = "Деканов Декан Деканович";
        string fname = "ФАИС";
        Fakultet target = new Fakultet(fileName, dekan, fname);
        string grupp = "ИТ-21";
        float expected = 6.95f;
        float actual;
        actual = target.SBall_Gruppy(grupp);
        Assert.AreEqual(expected, actual);
    }
```

Или еще один вариант:

```
public void SBall_GruppyTest()
  {
   string fileName = "D:\\test1.txt";
    string dekan = "Äåêàíîâ Äåêàí Äåêàíîâè÷";
    string fname = "ÔÀÈÑ";
     Fakultet target = new Fakultet(fileName,
dekan, fname);
string[] grupp = {"ÈÒ-21","ÈÒ-22","ÈÒ-23"};
float[] expected = {6.95f,7.5f,5.533333f};
string expectedstr = expected[0] + " " +
expected[1] + " " + expected[2]+" ";
 float[] actual=new float[3];
  string actualstr = "";
for (int i = 0; i < 3; i++)
            {
 actual[i] = target.SBall_Gruppy(grupp[i]);
 actualstr = actualstr + actual[i] + " ";
            }
  Assert.AreEqual(expectedstr, actualstr);

        }
```

На рисунке 4.4. показано окно с результатами прохождения теста.

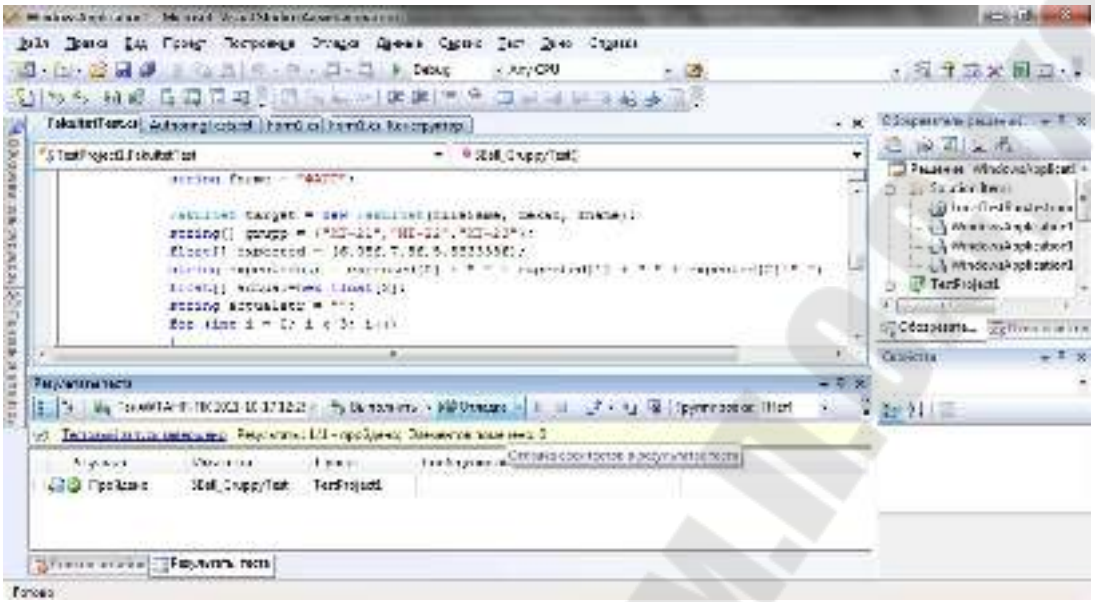


Рис. 4.4. Результаты прохождения теста

5. Разработка через тестирование

5.1. Основные понятия

TDD (Test Driven Development) – это разработка приложения через тестирование.

Разработка через тестирование - процесс разработки программного обеспечения, который предусматривает написание и автоматизацию модульных тестов еще до момента написания соответствующих классов или модулей.

- Это предсказуемый способ разработки программ. Разработчик знает, когда работу следует считать законченной, и можете не беспокоиться о длинной череде ошибок.
- У разработчика появляется шанс усвоить уроки, которые преподносит ему код. Если он воспользуется первой же идеей, которая пришла ему в голову, у него не будет шанса реализовать вторую, лучшую идею.
- Коллеги по команде могут рассчитывать на разработчика, а он, в, свою очередь, на них.
- Разработчику приятнее писать такой код.

Правила методики TDD:

- Пишем новый код только тогда, когда автоматический код не сработал.
- Удаляем дублирование.

Эти правила TDD определяют порядок этапов программирования:

- ***Красный*** – написать небольшой тест, который не работает, а возможно, даже не компилируется.
- ***Зеленый*** – заставить тест работать как можно быстрее, при этом не думать о правильности дизайна и чистоте кода. Написать ровно столько кода, чтобы тест сработал.
- ***Рефакторинг*** – удалить из написанного кода любое дублирование.

Другими словами, простая схема написания кода:

1. Пишем тест
2. Запускаем тест – он должен провалиться (программный код ведь еще не написан)
3. Пишем код, что бы тест заработал
4. Рефакторинг кода
5. Повторяем

Этот цикл повторяется снова и снова, пока не будет закончена работа над программным кодом.

Поскольку большинство фреймворков юнит-тестирования помечают неудавшиеся тесты красным цветом (например, выводится текст красного цвета), а удачный тест отмечается зеленым цветом (опять же выводится текст зеленого цвета), то данный цикл часто называют ***красным/зеленым циклом***.

Фреймворк *(англ. framework,* ***Каркас****) — структура программной системы; программное обеспечение, облегчающее разработку и объединение разных компонентов большого программного проекта.*

Идея проверять, что вновь написанный тест не проходит, помогает убедиться, что тест реально что-то проверяет. Только после этой проверки следует приступать к реализации новой функциональности.

Приём, известный как «красный/зеленый/рефакторинг», называют «мантрой разработки через тестирование». Под красным здесь понимают не прошедшие тесты, а под зелёным — прошедшие.

В TDD применяются все те же принципы, что были описаны в модульном тестировании: *написание тестов для небольших участков кода, тестирование в изоляции, автоматизация тестов.*

Преимущества:

- Программисты, использующие TDD на новых проектах, отмечают, что они реже ощущают необходимость использовать отладчик.
- Разработка через тестирование также влияет на дизайн программы. Изначально сфокусировавшись на тестах, проще представить, какая функциональность необходима пользователю. Таким образом, разработчик продумывает детали интерфейса до реализации.
- Несмотря на то, что при разработке через тестирование требуется написать большее количество кода, общее время, затраченное на разработку, обычно оказывается меньше. Тесты защищают от ошибок. Поэтому время, затрачиваемое на отладку, снижается многократно.

Слабые места:

- Главным недостатком TDD является то, что к нему сложно привыкнуть.
- Существуют задачи, которые невозможно (по крайней мере, на текущий момент) решить только при помощи тестов. В частности, TDD не позволяет механически продемонстрировать адекватность разработанного кода в области безопасности данных и взаимодействия между процессами.
- Требуется больше времени на разработку и поддержку. Если у организации нет уверенности в том, что разработка через тестирование улучшит качество продукта, то время, потраченное на написание тестов, может рассматриваться как потраченное впустую.
- Модульные тесты, создаваемые при разработке через тестирование, обычно пишутся теми же, кто пишет тестируемый код. Если разработчик неправильно истолковал требования к приложению, и тест, и тестируемый модуль будут содержать ошибку.
- Тесты сами по себе являются источником накладных расходов. Плохо написанные тесты, например, содержат жёстко вшитые строки с сообщениями об ошибках или подвержены ошибкам, дороги при поддержке. Чтобы упростить поддержку тестов следует повторно использовать сообщения об ошибках из тестируемого кода.

5.2. Порядок TDD разработки

Рассмотрим процесс разработки через тестирование на небольшом примере. Предположим, нужно разработать метод, вычисляющий корень квадратный числа, которое передается в метод в качестве входного параметра. Порядок разработки может быть следующим:

- Создать проект Visual C#. Данный проект будет содержать код, который необходимо предоставить.
- Создать тестовый проект. Для этого нужно выполнить команду меню

  Тест – Новый тест- Модульный тест
- Написать основной тестовый метод, внеся соответствующие изменения в созданный шаблон. Проверить результат, полученный для определенных входных данных:

```
[TestMethod]
  public void BasicRooterTest()
  {
     Rooter rooter = new Rooter();
    // Define a test input and output value:
    double expectedResult = 2.0;
    double input = expectedResult * expectedResult;
    // Run the method under test:
    double actualResult = rooter.SquareRoot(input);
    // Verify the result:
    Assert.AreEqual(expectedResult, actualResult);
  }
```

- Создать класс командой
  Проект – Добавить класс – задать имя **Rooter**

- В классе создать заглушку тестируемого метода
- Выполнить тест:

В меню **Тест** выбрать **Выполнение модульных тестов- Все тесты**.

Тест будет не пройден и выбросит соответствующее исключение.

На данном этапе создан тест и заглушка, которые будут изменены таким образом, что тест будет успешно пройден.

- В **Rooter.cs**, усовершенствуем код SquareRoot:

```
public double SquareRoot(double input)
{   return input / 2;  }
```

- Выполнить тест – тест пройден!
- Для уверенности, что код работает во всех случаях, добавить тесты, которые используют более широкий диапазон входных значений.

Совет: *Нужно избегать изменения существующих успешно выполненных тестов. Вместо этого нужно добавлять новые тесты. Изменять существующие тесты лучше только в тех случаях, когда пользовательские требования изменены. Эта политика позволяет не потерять существующие функциональные возможности при работе с расширенным кодом.*

- В тестовом классе добавить следующий тест, который использует диапазон входных значений:

```
[TestMethod]
public void RooterValueRange()
{
   Rooter rooter = new Rooter();
  for (double expectedResult = 1e-8
            expectedResult < 1e+8;
           expectedResult = expectedResult * 3.2)
    {
      RooterOneValue(rooter, expectedResult);
    }
}
private void RooterOneValue(Rooter rooter, double expectedResult)
  {
    double input = expectedResult * expectedResult;
    double actualResult = rooter.SquareRoot(input);
    Assert.AreEqual(expectedResult, actualResult);
  }
```

- Выполнить тесты. Новый тест завершается неудачей, несмотря на то, что по-прежнему первый тест завершается успешно.
- Изменить код тестируемого метода.

```
public double SquareRoot(double input)
  {
    double result = input;
    double previousResult = -input;
    while (Math.Abs(previousResult - result) > result / 1000)
    {
      previousResult = result;
     result = result - (result * result - input) / (2 *
 result);
    }
    return result;
  }
```

- Выполнить тест.
  Теперь оба теста успешно завершаются.
- Добавить тесты в исключительных случаях

```
[TestMethod]
        public void RooterTestNegativeInputx()
        {
            Rooter rooter = new Rooter();
            try
            {
                rooter.SquareRoot(-10);
            }
            catch (ArgumentOutOfRangeException e)
            {
                return;
            }
            Assert.Fail();
        }
```

- Чтобы этот тест проходил, нужно внести изменения в тестируемый метод:

```
public double SquareRoot(double input)
  {
    if (input <= 0.0)
    {
      throw new ArgumentOutOfRangeException();
    }
```

В заключение цитата из книги Кента Бека «Экстремальное программирование: разработка через тестирование».

«Сравним программирование с подъемом ведра из колодца. Ведро наполнено водой, вы вращаете рычаг, наматывая тем самым цепь на барабан и поднимая ведро вверх. Если ведро небольшое, вполне подойдет обычный рычаг, напрямую соединенный с барабаном. Но если ведро большое и тяжелое, вы наверняка устанете, пока будете поднимать его на самый верх. Чтобы обеспечить возможность отдыхать между поворотами рычага, вам потребуется храповой механизм, который позволит фиксировать рычаг в случае, если вы устали. Чем тяжелее ведро, тем ближе друг к другу должны располагаться шипы на шестеренке храповика.

Тесты в TDD - это шипы на шестеренке храповика. Заставив тест работать, мы знаем, что теперь тест работает, отныне и навеки. Мы стали на шаг ближе к завершению работы, чем мы были до того, как тест сработал. После этого мы заставляем второй тест работать, затем третий, четвертый и т.д. Чем сложнее проблема, стоящая перед программистом, тем меньшую область функциональности должен покрывать каждый тест….

TDD – это набор способов, ведущих к простым программным решениям, которые может применять любой разработчик, а также тестов, придающих уверенность в работе. Если вы гений, эти способы вам не нужны. Если вы тугодум – они вам все равно не помогут. Для всех же нас, кто находится между этими крайностями, следование двум простым правилам поможет работать намного эффективнее» .

Литература

1. Бек К. Экстремальное программирование: разработка через тестирование. Библиотека программиста. – СПб.:Питер,2003. – 224с.
2. Канер Сэм и др. Тестирование программного обеспечения. Фундаментальные концепции менеджмента бизнес-приложений: Пер. с англ./Сэм Канер, Джек Фолк, Енг Кек Нгуен. — К.: Издательство «ДиаСофт», 2001. — 544 с.
3. Котляров В.П. Основы тестирования программного обеспечения: Учебное пособие / В.П.Котляров, Т.В.Коликова – М.:Интернет-Университет Информационных Технологий; БИНОМ. Лаборатория знаний, 2006.-285с.
4. Павловская Т.А. C#. Программирование на языке высокого уровня: Учебник для вузов. – СПб.: Питер, 2007. – 432с.
5. Савин Р. Тестирование Дот Ком, или Пособие по жестокому обращению с багами в интернет-стартапах. — М.: Дело, 2007. — 312 с.
6. Синицын С.В., Налютин Н.Ю. Верификация программного обеспечения: Учебное пособие. М.: Интернет-Университет Информационных Технологий; БИНОМ. Лаборатория знаний, 2008. - 368 с
7. Тамре Л. Введение в тестирование программного обеспечения: Пер. с англ. — М.: Издательский дом "Вильямс", 2003. — 368 с

СОДЕРЖАНИЕ

ТЕСТИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

КУРС ЛЕКЦИЙ
по одноименной дисциплине
для слушателей специальности 1-40 01 73
«Программное обеспечение
информационных систем»
заочной формы обучения

Составитель: **Романькова** Татьяна Леонидовна

Подписано к размещению в электронную библиотеку
ГГТУ им. П. О. Сухого в качестве электронного
учебно-методического документа 21.05.14.
Рег. № 79Е.
http://www.gstu.by