Государственное бюджетное профессиональное образовательное учреждение
«Кунгурский автотранспортный колледж»

МЕТОДИЧЕСКИЕ РЕКОМЕНДАЦИИ ПО ВЫПОЛНЕНИЮ ПРАКТИЧЕСКИХ РАБОТ

по междисциплинарному курсу

МДК.01.02 Поддержка и тестирование программных модулей

по специальности:

09.02.07 Информационные системы и программирование

(код и наименование специальности)

2020

Одобрено на заседании цикловой комиссии информационно-математических дисциплин
Протокол №____ от «__» _____20___г.
Председатель комиссии
_____Е.А. Наговицына

УТВЕРЖДАЮ
Зам. директора
_____М.Г. Целищева
«___» _____ 20___г.

Организация-разработчик: ГБПОУ КАТК

СОДЕРЖАНИЕ

1 Пояснительная записка

Данные методические рекомендации составлены в соответствии с содержанием рабочей программы МДК.01.02 Поддержка и тестирование программных модулей специальности 09.02.07 Информационные системы и программирование.

МДК.01.02 Поддержка и тестирование программных модулей изучается в течение IV семестра. Общий объем времени, отведенный на практические занятия по УД, составляет в соответствии с учебным планом и рабочей программой – 56 часов

Практические работы проводятся после изучения соответствующих разделов и тем МДК.01.02 Поддержка и тестирование программных модулей. Выполнение обучающимися практических работ позволяет им понять, где и когда изучаемые теоретические положения и практические умения могут быть использованы в будущей практической деятельности.

В результате выполнения практических работ, предусмотренных программой по МДК.01.02 Поддержка и тестирование программных модулей, обучающийся должен:
уметь:
- выполнять отладку и тестирование программы на уровне модуля;
- осуществлять разработку кода программного модуля на современных языках программирования;
- уметь выполнять оптимизацию и рефакторинг программного кода;

знать:
- способы оптимизации и приемы рефакторинга;
- основные принципы отладки и тестирования программных продуктов;

Вышеперечисленные умения, знания и практический опыт направлены на формирование следующих профессиональных и общих компетенций обучающихся**:**
ПК.1.3. Выполнять отладку программных модулей с использованием специализированных программных средств
ПК 1.4. Выполнять тестирование программных модулей
ПК 1.5. Осуществлять рефакторинг и оптимизацию программного кода
ОК 1. Выбирать способы решения задач профессиональной деятельности, применительно к различным контекстам
ОК 2. Осуществлять поиск, анализ и интерпретацию информации, необходимой для выполнения задач профессиональной деятельности
ОК 4. Работать в коллективе и команде, эффективно взаимодействовать с коллегами, руководством, клиентами.
ОК 5. Осуществлять устную и письменную коммуникацию на государственном языке с учетом особенностей социального и культурного контекста.
ОК 9. Использовать информационные технологии в профессиональной деятельности
ОК10. Пользоваться профессиональной документацией на государственном и иностранном языках

2 Перечень практических работ УД МДК.01.02 Поддержка и тестирование программных модулей

Название практических работ	**Количество часов**
Тема 1.2.1 Отладка и тестирование программного обеспечения	
Практическая работа №1 Тестирование «белым ящиком»	8
Практическая работа №2 Тестирование «черным ящиком»	8
Практическая работа №3 Модульное тестирование	8
Практическая работа №4 Интеграционное тестирование	8

Тема 1.2.2Документирование	
Практическая работа №5 Оформление документации на программные средства с использованием инструментальных средств.	24
Итого: 56 часов	

3 Инструктивно-методические указания по выполнению практических работ

Практическая работа №1 Тестирование «белым ящиком»

Цель работы: изучить методы тестирования логики программы, формализованные описания результатов тестирования и стандарты по составлению схем программ.

Теоретическая часть. Виды тестирования

Тестирование программного обеспечения включает в себя целый комплекс действий, аналогичных последовательности процессов разработки программного обеспечения. В него входят :

- постановка задачи для теста;
- проектирование теста;
- написание тестов;
- тестирование тестов;
- выполнение тестов;
- изучение результатов тестирования.

Наиболее важным является проектирование тестов. Существуют разные подходы к проектированию тестов.

Первый состоит в том, что тесты проектируются на основе внешних спецификаций программ и модулей либо спецификаций сопряжения модуля с другими модулями, программа при этом рассматривается как «черный ящик». Смысл теста заключается в том, чтобы проверить, соответствует ли программа внешним спецификациям. При этом содержание модуля не имеет значения. Такой подход получил название — стратегия «черного ящика».

Второй подход — стратегия «белого ящика», основан на анализе логики программы. При таком подходе тестирование заключается в проверке каждого пути, каждой ветви алгоритма. При этом внешняя спецификация во внимание не принимается.

Ни один из этих подходов не является оптимальным. Реализация тестирования методом «черного ящика» сводится к проверке всех возможных комбинаций входных данных. Невозможно протестировать программу, подавая на вход бесконечное множество значений, поэтому ограничиваются определенным набором данных. При этом исходят из максимальной отдачи теста по сравнению с затратами на его создание. Она измеряется вероятностью того, что тест выявит ошибки, если они имеются в программе. Затраты измеряются временем и стоимостью подготовки, выполнения и проверки результатов теста.

Тестирование методом «белого ящика» также не дает 100%-ной гарантии того, что модуль не содержит ошибок. Даже если предположить, что выполнены тесты для всех ветвей алгоритма, нельзя с полной уверенностью утверждать, что программа соответствует ее спецификациям. Например, если требовалось написать программу для вычисления кубического корня, а программа фактически вычисляет корень квадратный, то реализация будет совершенно неправильной, даже если проверить все пути. Вторая проблема — отсутствующие пути. Если программа реализует спецификации не полностью (например, отсутствует такая специализированная функция, как проверка на отрицательное значение входных данных программы вычисления квадратного корня), никакое тестирование существующих путей не выявит такой ошибки. И наконец, проблема зависимости результатов тестирования от входных данных. Одни данные будут давать правильные результаты, а другие нет. Например, если для определения равенства трех чисел программируется выражение вида:

IF *(A* + *B*+ C)/3 = Д

то оно будет верным не для всех значений *A, B* и *C* (ошибка возникает в том случае, когда из двух значений *B* или C одно больше, а другое на столько же меньше *A)*. Если концентрировать внимание только на тестировании путей, нет гарантии, что эта ошибка будет выявлена.

Таким образом, полное тестирование программы невозможно, т. е. никакое тестирование не гарантирует полное отсутствие ошибок в программе. Поэтому необходимо проектировать тесты таким образом, чтобы увеличить вероятность обнаружения ошибки в программе.

Стратегия «белого ящика»

Существуют следующие методы тестирования по принципу «белого ящика»:

- покрытие операторов;
- покрытие решений;
- покрытие условий;
- покрытие решений/условий;
- комбинаторное покрытие условий.

Метод покрытия операторов

Целью этого метода тестирования является выполнение каждого оператора программы хотя бы один раз.

Пример.

Если для тестирования задать значения переменных *A = 2,* B = 0, *X=3,* будет реализован путь *ace,* т. е. каждый оператор программы выполнится один раз (рис. Л5.1, *а).* Но если внести в алгоритм ошибки — заменить в первом условии *and* на *or,* а во втором *X*> 1 на *X*< 1 (рис. Л5.1, *б),* ни одна ошибка не будет обнаружена (табл. Л5.1). Кроме того, путь *abd* вообще не будет охвачен тестом, и если в нем есть ошибка, она также не будет обнаружена. В табл. Л5.1 ожидаемый результат определяется по блок-схеме на рис. Л5.1, *а,* а фактический — по рис. Л5.1, *б.*

Как видно из этой таблицы, ни одна из внесенных в алгоритм ошибок не будет обнаружена.

Таблица Л5.1. **Результат тестирования** методом **покрытия операторов**

Тест	Ожидаемый результат	Фактический результат	Результат тестирования
A = 2, B = 0, X=3	*X=2,5*	*X=2,5*	Неуспешно

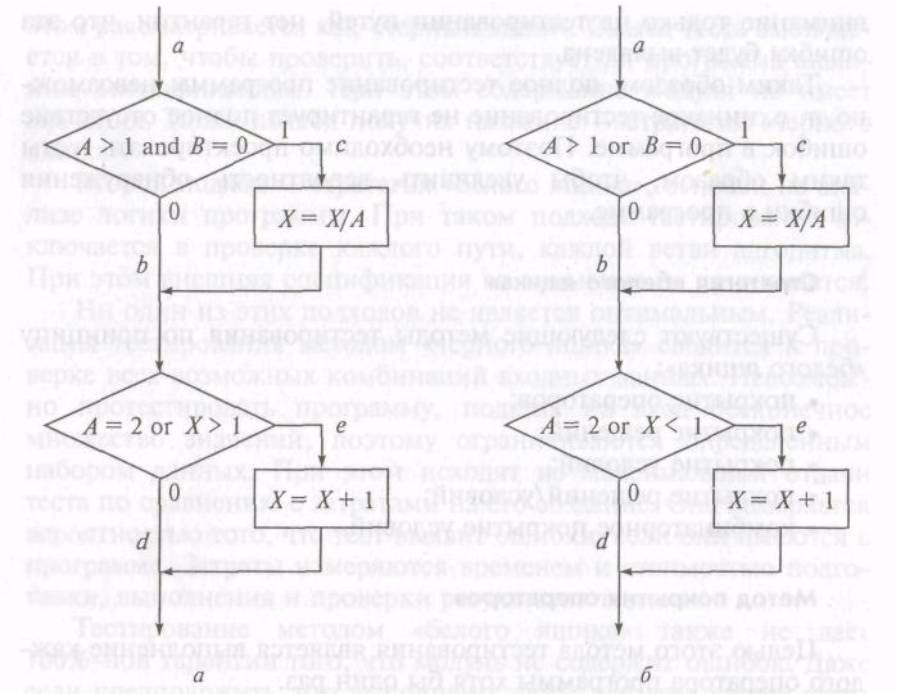


Рис. Л5.1. Пример алгоритма программы:
а — правильный; *б* — с ошибкой

Метод покрытия решений (покрытия переходов)

Согласно методу покрытия решений каждое направление перехода должно быть реализовано, по крайней мере, один раз. Этот метод включает в себя критерий покрытия операторов, так как при выполнении всех направлений переходов выполнятся все операторы, находящиеся на этих направлениях.

Для программы, приведенной на рис. Л5.1, покрытие решений может быть выполнено двумя тестами, покрывающими пути *{ace, abd*, либо *{acd, abe]*. Для этого выберем следующие исходные данные: {A = 3, B=0, *X=3}* — в первом случае и *{A = 2, B=1, X=* 1} — во втором. Однако путь, где А не меняется, будет проверен с вероятностью 50%: если во втором условии вместо условия *X*> 1 записано *X*< 1, то ошибка не будет обнаружена двумя тестами.

Результаты тестирования приведены в табл. Л5.2.

Таблица Л5.2. Результат тестирования методом покрытия решений

Тест	Ожидаемый результат	Фактический результат	Результат тестирования
A = 3, B=0, X=3	*X=1*	*X=1*	Неуспешно
A = 2, B=1,X=1	*X=1*	*X=1,5*	Успешно

Метод покрытия условий

Этот метод может дать лучшие результаты по сравнению с предыдущими. В соответствии с методом покрытия условий записывается число тестов, достаточное для того, чтобы все возможные результаты каждого условия в решении выполнялись, .по крайней мере, один раз.

В рассматриваемом примере имеем условия: *{A*> 1, *B = 0}, {A* = 2, *X*> 1}. Следовательно, требуется достаточное число тестов, такое, чтобы реализовать ситуации, где *A* > 1, *A* < 1, *B*= О и *B*<> 0 в точке *а* и *A* = 2, *A*<> 2, *X*> 1 и *X*< 1 в точке *b.* Тесты, удовлетворяющие критерию покрытия условий (табл. Л5.3), и соответствующие им пути:

а) *A* = 2, B=0, *X=4 ace;*

б)A= 1, *B* = 1, *X=0 abd.*

Таблица Л5.3. Результаты тестирования методом покрытия условий

Тест	Ожидаемый результат	Фактический результат	Результат тестировани
A = 2, B=0, *X*= 4	X=3	X=3	Неуспешно
A=1, B=1,X=0	X=0	*X*=1	Успешно

Метод покрытия решений/условий

Критерий покрытия решений/условий требует такого достаточного набора тестов, чтобы все возможные результаты каждого условия выполнялись по крайней мере один раз, все результаты каждого решения выполнялись по крайней мере один раз и, кроме того, каждой точке входа передавалось управление по крайней мере один раз.

Недостатки метода:

- не всегда можно проверить все условия;
- невозможно проверить условия, которые скрыты другими условиями;
- • недостаточная чувствительность к ошибкам в логических выражениях.
- Так, в рассматриваемом примере два теста метода покрытия условий
- а) *A = 2,* B=0, *X=4 ace;*
- б)A= 1, *B=1,X=0 abd*
- отвечают и критерию покрытия решений/условий. Это является следствием того, что одни условия приведенных решений скрывают другие условия в этих решениях. Так, если условие *A* > 1 будет ложным, транслятор может не проверять условия *B=0,* поскольку при любом результате условия ,B=0 результат решения *((A*> 1)&(B=0)) примет значение *ложь*. То есть в варианте на рис. Л5.1 не все результаты всех условий выполнятся в процессе тестирования.
- Рассмотрим реализацию того же примера на рис. Л5.2. Наиболее полное покрытие тестами в этом случае осуществляется так, чтобы выполнялись все возможные результаты каждого простого решения. Для этого нужно покрыть пути *aceg* (тест *A = 2*, B=0, X=4), *acdfh* (тест A = 3, B= 1,X=0), *abfh* (тест A = 0, B = 0, *X*= 0), *abfi* (тест *A* = 0, B= 0, *X*= 2).

+

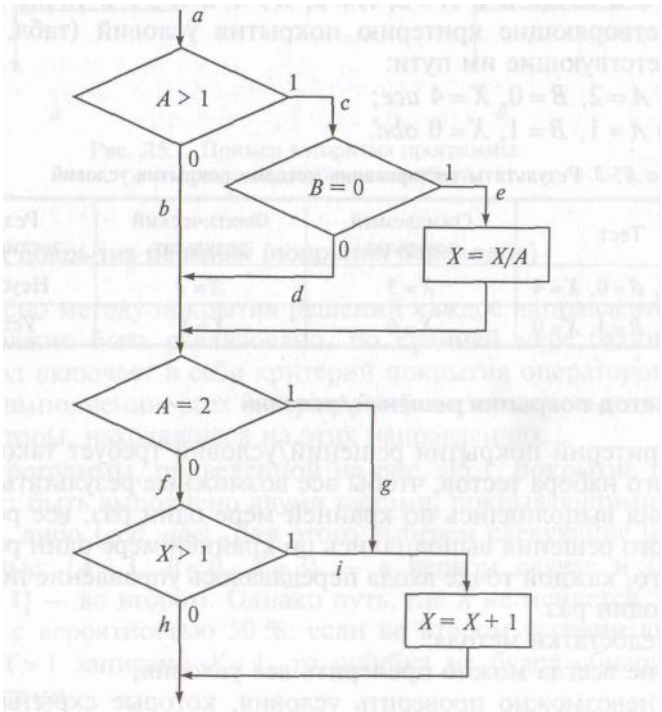
- 

Рис. Л5.2. Пример алгоритма программы
- ○

-

Протестировав алгоритм на рис. Л5.2, нетрудно убедиться в том, что критерии покрытия условий и критерии покрытия решений/условий недостаточно чувствительны к ошибкам в логических выражениях.

- **Метод комбинаторного покрытия условий**
- Критерий комбинаторного покрытия условий удовлетворяет также и критериям покрытия решений, покрытия условий и покрытия решений/условий.
- Этот метод требует создания такого числа тестов, чтобы все возможные комбинации результатов условия в каждом решении выполнялись по крайней мере один раз. По этому критерию в рассматриваемом примере должны быть покрыты тестами следующие восемь комбинаций:
  -
  -

1.A>1,B=0. *5.A = 2,X>1.*

- 2. A > 1,B<>0. *6.A = 2,X<1.*
- 3.A<1,B=0. *7.A<>2,X>1.*
- 4.A<1,B<>0. 8.A<>2,X<1.
- Для того чтобы протестировать эти комбинации, необязательно использовать все 8 тестов. Фактически они могут быть покрыты четырьмя тестами (табл. JI5.4):
- *A = 2,* B= 0, *X*= 4 {покрывает 1, 5};
- *A* = 2, B= 1, *X*= 1 {покрывает 2, 6};
- *A* = 0,5, B = 0, *X= 2* {покрывает 3, 7};
-
- *A*=1,B=0, *X*= 1 {покрывает 4, 8}.
- *Таблица Л5.4.*
- Результаты тестирования методом комбинаторного покрытия условий

Тест	Ожидаемый результат	Фактический результат	Результат тестирования
A = 2, *B=0,X=4*	*X=3*	*X=3*	Неуспешно
A= 2, *B=1,X=1*	*X=2*	*X*= 1,5	Успешно
A = 0,5, b=0,x=2	*X=3*	*X=4*	Успешно
A=1, B=0,X= 1	*X=1*	*X=1*	Неуспешно

- **Функциональное тестирование**или тестирование по методом четного ящика базируется на том, что все тесты основаны на спецификации системы или ее компонентов. Поведение системы определяется изучением ее входных и соответствующих выходных данных.
- При тестировании «черного ящика» тестировщик имеет доступ к ПО только через те же интерфейсы, что и заказчик или пользователь, либо через внешние интерфейсы, позволяющие другому компьютеру либо другому процессу подключиться к системе для тестирования. Например, тестирующий модуль может виртуально нажимать клавиши или кнопки мыши в тестируемой программе с помощью механизма взаимодействия процессов с уверенностью в том, что эти события вызывают тот же отклик, что и реальные нажатия клавиш и кнопок мыши.
- ***Порядок выполнения работы***

1. Спроектировать тесты по принципу «белого ящика» для программы, разработанной в лабораторной работе № 4. Использовать схемы алгоритмов, разработанные и уточненные в лабораторных работах № 2, 3.
2. Выбрать несколько алгоритмов для тестирования и обо значить буквами или цифрами ветви этих алгоритмов.
3. Выписать пути алгоритма, которые должны быть проверены тестами для выбранного метода тестирования.
4. Записать тесты, которые позволят пройти по путям алгоритма.
5. Протестировать разработанную вами программу. Результаты оформить в виде таблиц (см. табл. Л5.1— Л5.4).
6. Проверить все виды тестов и сделать выводы об их эффективности.
7. Оформить отчет по лабораторной работе.
8. Сдать и защитить работу.

+

- ***Защита отчета по лабораторной работе***
- Отчет по лабораторной работе должен состоять из:

1. Постановки задачи.
2. Блок-схемы программ.
3. Тестов.
4. Таблиц тестирования программы.
5. Выводов по результатам тестирования (не забывайте, что целью тестирования является обнаружение ошибок в программе).

Практическая работа №2 Тестирование «черным ящиком»

Методология составления тестов "чёрного ящика"

- Эквивалентное разбиение
- Анализ граничных значений
- Применение функциональных диаграмм
- Предположение об ошибке

- Эквивалентное разбиение

- Тестирование программы ограничивается использованием небольшого подмножества всех возможных входных данных. Тогда, конечно, хотелось бы выбрать для тестирования самое подходящее подмножество (т. с. подмножество с наивысшей вероятностью обнаружения большинства ошибок). Правильно выбранный тест этого подмножества должен обладать двумя свойствами:

- уменьшать, причем более чем на единицу, число других тестов, которые должны быть разработаны для достижения заранее определенной цели "приемлемого" тестирования;

- покрывать значительную часть других возможных тестов, что в некоторой степени свидетельствует о наличии или отсутствии ошибок до и после применения этого ограниченного множества значений входных данных.

- Указанные свойства, несмотря на их кажущееся подобие, описывают два различных положения. Во-первых, каждый тест должен включать столько различных входных условий, сколько это возможно, с тем, чтобы минимизировать общее число необходимых тестов. Во-вторых, необходимо пытаться разбить входную область программы на конечное число классов эквивалентности так, чтобы можно было предположить (конечно, не абсолютно уверенно), что каждый тест, являющийся представителем некоторого класса, эквивалентен любому другому тесту этого класса. Иными словами, если один тест класса эквивалентности обнаруживает ошибку, то следует ожидать, что и все другие тесты этого класса эквивалентности будут обнаруживать ту же самую ошибку. Наоборот, если тест не обнаруживает ошибки, то следует ожидать, что ни один тест этого класса эквивалентности не будет обнаруживать ошибки (в том случае, когда некоторое подмножество класса эквивалентности не попадает в пределы любого другого класса эквивалентности, так как классы эквивалентности могут пересекаться). Эти два положения составляют основу методологии тестирования по принципу черного ящика, известной как эквивалентное разбиение. Второе положение используется для разработки набора "интересных" условий, которые должны быть протестированы, а первое - для разработки минимального набора тестов, покрывающих эти условия. Примером класса эквивалентности для программы о треугольнике является набор "трех равных чисел, имеющих целые значения, большие нуля". Определяя этот набор как класс эквивалентности, устанавливают, что если ошибка не обнаружена некоторым тестом данного набора, то маловероятно, что она будет обнаружена другим тестом набора. Иными словами, в этом случае время тестирования лучше затратить на что-нибудь другое (на тестирование других классов эквивалентности). Разработка тестов методом эквивалентного разбиения осуществляется в два этапа:

1. выделение классов эквивалентности
2. построение тестов.

- Выделение классов эквивалентности

- Классы эквивалентности выделяются путем выбора каждого входного условия (обычно это предложение или фраза в спецификации) и разбиением его на две или более групп.

- Заметим, что различают два типа классов эквивалентности: *правильные классы*эквивалентности, представляющие правильные входные данные программы, и*неправильные классы эквивалентности*, представляющие все другие возможные состояния условий (т. е. ошибочные входные значения). При этом существует ряд правил:

+

1. Если входное условие описывает область значений (например, "целое данное может принимать значения от 1 до 999"), то определяются один правильный класс эквивалентности (1 <= значение целого данного <= 999) и два неправильных (значение целого данного < 1 и значение целого данного > 999).

2. Если входное условие описывает множество входных значений и есть основание полагать, что каждое значение программа трактует особо (например, "известны способы передвижения на АВТОБУСЕ, ГРУЗОВИКЕ, ТАКСИ, ПЕШКОМ или МОТОЦИКЛЕ"), то определяется правильный класс эквивалентности для каждого значения и один неправильный класс эквивалентности (например, "НА ПРИЦЕПЕ").

3. Если входное условие описывает ситуацию "должно быть" (например, "первым символом идентификатора должна быть буква"), то определяется один правильный класс эквивалентности (первый символ - буква) и один неправильный (первый символ - не буква).

4. Если есть любое основание считать, что различные элементы класса эквивалентности трактуются программой неодинаково, то данный класс эквивалентности разбивается на меньшие классы эквивалентности.

- Построение тестов
- Второй шаг заключается в использовании классов эквивалентности для построения тестов. Этот процесс включает в себя:

+

1. Назначение каждому классу эквивалентности уникального номера.

2. Проектирование новых тестов, каждый из которых покрывает как можно большее число непокрытых правильных классов эквивалентности, до тех пор, пока все правильные классы эквивалентности не будут покрыты (только не общими) тестами.

3. Запись тестов, каждый из которых покрывает один и только один из непокрытых неправильных классов эквивалентности, до тех пор, пока все неправильные классы эквивалентности не будут покрыты тестами. Причина покрытия неправильных классов эквивалентности индивидуальными тестами состоит в том, что определенные проверки с ошибочными входами скрывают или заменяют другие проверки с ошибочными входами. Например, спецификация устанавливает "тип книги при поиске (ВЫЧИСЛИТЕЛЬНАЯ ТЕХНИКА, ПРОГРАММИРОВАНИЕ или ОБЩИЙ) и количество (1-9999)". Тогда тест XYZ О отображает два ошибочных условия (неправильный тип книги и количество) и, вероятно, не будет осуществлять проверку количества, так как программа может ответить: "XYZ-НЕСУЩЕСТВУЮЩИЙ ТИП КНИГИ" и не проверять остальную часть входных данных.

- **<u>Пример</u>**

- Предположим, что при разработке компилятора для подмножества языка программирования требуется протестировать синтаксическую проверку оператора DIM[l].
- Пусть определена следующая спецификация:
- Оператор DIM используется для определения массивов.
- DIM *ad* [,*ad*]…,где*ad* есть описатель массива в форме*n*(*d*[,*d*]...),*n* – символическое имя массива, а*d* – индекс массива.
- Символические имена могут содержать от одного до шести символов - букв или цифр, причем первой должна быть буква.
- Допускается от одного до семи индексов. Форма индекса [lb:] ub, где lb и ub задают нижнюю и верхнюю границы индекса массива. Граница может быть либо константой, принимающей значения от - 65534 до 65535, либо целой переменной (без индексов). Если lb не определена, то предполагается, что она равна единице. Значение ub должно быть больше или равно lb. Если lb определена, то она может

иметь отрицательное, нулевое или положительное значение. Оператор модет распологатьсяна нескольких строках (Конец спецификации.)

- Первый шаг заключается в том, чтобы идентифицировать входные условия и по ним определить классы эквивалентности. Классы эквивалентности в таблице обозначены числами.

Входные условия	Правильные классы эквивалентности	Неправильные классы эквивалентности
Число описателей массивов	ОДИН (1), > ОДНОГО (2)	НИ ОДНОГО (3)
Длина имени массива	1-6(4)	0(5), > 6(6)
Имя массива	Имеет в своем составе буквы (7) и цифры (8)	содержит что-то еще (9)
Имя массива начинается с буквы	да (10)	нет (11)
Число индексов	1-7(12)	0(13), > 7(14)
Верхняя граница	Константа (15), целая переменная (16)	имя элемента массива (17), что-то иное (18)
Имя целой переменной	Имеет в своем составе буквы (19), и цифры (20)	состоит из чего-то еще (21)
Целая переменная начинается с буквы	да (22)	нет (23)
Константа	От -65534 до 65535 (24))	Меньше -65534 (25), больше 65535 (26)
Нижняя граница определена	да (27), нет (28)	
Верхняя граница по отношению к нижней границе	Больше (29), равна (30)	меньше (31)
Значение нижней границы	Отрицательное (32) Нуль (33), > 0 (34)	
Нижняя граница	Константа (35), целая переменная	имя элемента массива (37), что-то

	(36)	иное (38)
○ Оператор расположен на нескольких строках	○ да (39), нет (40)	○

- Следующий шаг - построение теста, покрывающего один или более правильных классов эквивалентности.
- Например, тест DIM A(2) покрывает классы 1, 4, 7, 10, 12, 15, 24, 28, 29 и 40.
- Далее определяются один или более тестов, покрывающих оставшиеся правильные классы эквивалентности.
- Так, тест
- DIM A12345(I,9,J4XXXX.65535,1,KLM, X 100),
- BBB (-65534 : 100,0 : 1000,10 : 10,1 : 65535)
- покрывает оставшиеся классы.
- Перечислим неправильные классы эквивалентности и соответствующие им тесты:

○ (3)	○ DIMENSION
○ (5)	○ DIMENSION(10)
○ ((6)	○ DIMENSION A234567(2)
○ (9)	○ DIMENSION A.I(2)
○ (11)	○ DIMENSION1A(10)
○ (13)	○ DIMENSION B
○ (14)	○ DIMENSION B (4,4,4,4,4,4,4,4)
○ (17)	○ DIMENSION B(4,A(2))
○ (18)	○ DIMENSION B(4,,7)
○ (21)	○ DIMENSION C(1.,10)
○ (23)	○ DIMENSION C(10,1J)
○ (25)	○ DIMENSION D (-65535:1)
○ (26)	○ DIMENSION D(65536)
○ (31)	○ DIMENSION D(4:3)
○ (37)	○ DIMENSION D(A(2):4)
○ (38)	○ DIMENSION D(.:4)

- Эти классы эквивалентности покрываются 18 тестами.
- Хотя эквивалентное разбиение значительно лучше случайного выбора тестов, оно все же имеет недостатки (т. е. пропускает определенные типы высокоэффективных тестов).

- Следующие два метода - анализ граничных значений и использование функциональных диаграмм (диаграмм причинно-следственных связей cause-effect graphing) - свободны от многих недостатков, присущих эквивалентному разбиению.
-
    - Анализ граничных значений
- Как показывает опыт, тесты, исследующие граничные условия, приносят большую пользу, чем тесты, которые их не исследуют. Граничные условия - это ситуации, возникающие непосредственно на, выше или ниже границ входных и выходных классов эквивалентности. Анализ граничных значений отличается от эквивалентного разбиения в двух отношениях:

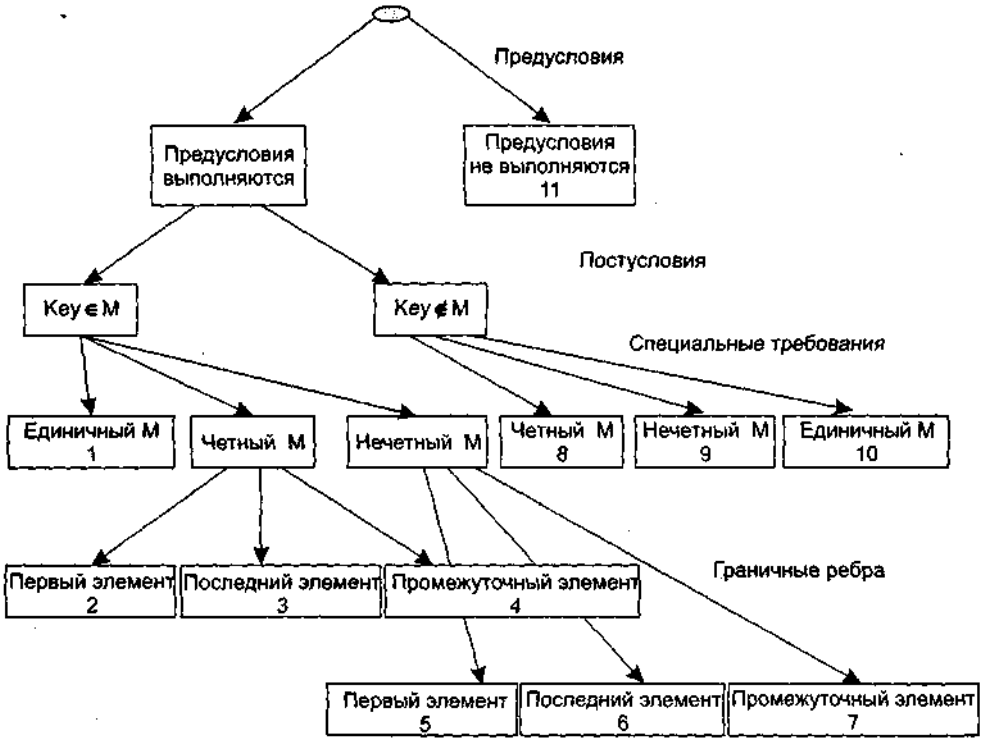
1. Выбор любого элемента в классе эквивалентности в качестве представительного при анализе граничных значений осуществляется таким образом, чтобы проверить тестом каждую границу этого класса.
2. При разработке тестов рассматривают не только входные условия (пространство входов), но и пространство результатов (т. е. выходные классы эквивалентности) .

- Приведем несколько общих правил этого метода.
  +

1. Построить тесты для границ области и тесты с неправильными входными данными для ситуаций незначительного выхода за границы области, если входное условие описывает область значений. Например, если правильная область входных значений есть - 1,0 - +1,0, то написать тесты для ситуаций - 1,0, 1,0, - 1,001 и 1,001.
2. Построить тесты для минимального и максимального значений условий и тесты, большие и меньшие этих значений, если входное условие удовлетворяет дискретному ряду значений. Например, если входной файл может содержать от 1 до 255 записей, то получить тесты для 0,1, 255 и 256 записей.
3. Использовать правило 1 для каждого выходного условия. Например, если программа вычисляет ежемесячный расход и если минимум расхода составляет 0,00 дол., а максимум - 1165,25 дол., то построить тесты, которые вызывают расходы с 0,00 дол. и 1165,25 дол. Кроме того, построить, если это возможно, тесты, которые вызывают отрицательный расход и расход больше 1165,25 дол. Заметим, что важно проверить границы пространства результатов, поскольку не всегда границы входных областей представляют такой же набор условий, как и границы выходных областей (например, при рассмотрении подпрограммы вычисления синуса). Не всегда также можно получить результат вне выходной области, но, тем не менее, стоит рассмотреть эту возможность.
4. Использовать правило 2 для каждого выходного условия. Например, если система информационного поиска отображает на экране терминала наиболее релевантные рефераты в зависимости от входного запроса, но никак не более четырех рефератов, то построить тесты, такие, чтобы программа отображала нуль, один и четыре реферата, и тест, который мог бы вызвать выполнение программы с ошибочным отображением пяти рефератов.
5. Если вход или выход программы есть упорядоченное множество (например, последовательный файл, линейный список, таблица), то сосредоточить внимание на первом и последнем элементах этого множества.
6. Существенное различие между анализом граничных значений и эквивалентным разбиением заключается в том, что анализ граничных значений исследует ситуации, возникающие на и вблизи границ эквивалентных разбиений.
7. Пример
8. Положим, что нужно протестировать программу бинарного поиска. Нам известна *спецификация* этой программы. Поиск выполняется в массиве

элементов *M,* возвращается индекс *I* элемента массива, значение которого соответствует ключу поиска *Key.*

9. *Входные условия:*
10. 1) массив должен быть упорядочен;
11. 2) массив должен иметь не менее одного элемента;
12. 3) нижняя граница массива (индекс) должна быть меньше или равна его верхней границе.
13. *Результаты::*
14. 1) если элемент найден, то флаг Result=True, значение I — номер элемента;
15. 2) если элемент не найден, то флаг Result=False, значение I не определено.
16. Для формирования классов эквивалентности (и их ребер) надо построить дерево разбиений. Листья дерева разбиений дадут нам искомые классы эквивалентности. Определим стратегию разбиения. На первом уровне будем анализировать выполнимость входных условий, на втором уровне — выполнимость результатов. На третьем уровне можно анализировать специальные требования, полученные из практики разработчика. В нашем примере мы знаем, что входной массив должен быть упорядочен. Обработка упорядоченных наборов из четного и нечетного количества элементов может выполняться по-разному. Кроме того, принято выделять специальный случай одноэлементного массива. Следовательно, на уровне специальных требований возможны следующие эквивалентные разбиения:
17. 1) массив из одного элемента;
18. 2) массив из четного количества элементов;
19. 3) массив из нечетного количества элементов, большего единицы.
20. Наконец на последнем, 4-м уровне критерием разбиения может быть анализ ребер классов эквивалентности. Очевидно, возможны следующие варианты:
21. 1) работа с первым элементом массива;
22. 2) работа с последним элементом массива;
23. 3) работа с промежуточным (ни с первым, ни с последним) элементом массива.
24. Структура дерева разбиений приведена на рис. 7.3.



25.
26. Дерево разбиений области исходных данных бинарного поиска
27.

28. Это дерево имеет 11 листьев. Каждый лист задает отдельный тестовый вариант. Покажем тестовые варианты, основанные на проведенных разбиениях.
29. *Тестовый вариант 1 (единичный массив, элемент найден)***:**
30. *ИД:* M=15; Key=15.
31. *ОЖ.РЕЗ.:* Resutt=True; I=1.
32. *Тестовый вариант 2 (четный массив, найден 1-й элемент)***:**
33. *ИД:* M=15, 20, 25,30,35,40; Key=15.
34. *ОЖ.РЕЗ.:* Result=True; I=1.
35. *Тестовый вариант 3 (четный массив, найден последний элемент)* **:**
36. *ИД:* M=15, 20, 25, 30, 35, 40; Key=40.
37. *ОЖ.РЕЗ:*. Result=True; I=6.
38. *Тестовый вариант 4 (четный массив, найден промежуточный элемент)***:**
39. *ИД:* M=15,20,25,30,35,40; Key=25.
40. *ОЖ.РЕЗ.:* Result-True; I=3.
41. *Тестовый вариант 5 (нечетный массив, найден 1-й элемент)***:**
42. *ИД:* M=15, 20, 25, 30, 35,40, 45; Key=15.
43. *ОЖ.РЕЗ.:* Result=True; I=1.
44. *Тестовый вариант 6 (нечетный массив, найден последний элемент)***:**
45. *ИД:* M=15, 20, 25, 30,35, 40,45; Key=45.
46. *ОЖ.РЕЗ.:* Result=True; I=7.
47. *Тестовый вариант 7 (нечетный массив, найден промежуточный элемент)***:**
48. *ИД:* M=15, 20, 25, 30,35, 40, 45; Key=30.
49. *ОЖ.РЕЗ.:* Result=True; I=4.
50. *Тестовый вариант 8 (четный массив, не найден элемент)***:**
51. *ИД:* M=15, 20, 25, 30, 35,40; Key=23.
52. *ОЖ.РЕЗ.:* Result=False; I=?
53. *Тестовый вариант 9 (нечетный массив, не найден элемент)***;**
54. *ИД:* M=15, 20, 25, 30, 35, 40, 45; Key=24.
55. *ОЖ.РЕЗ:*. Result=False; I=?
56. *Тестовый вариант 10 (единичный массив, не найден элемент)***:**
57. *ИД:* M=15; Key=0.
58. *ОЖ.РЕЗ.:* Result=False; I=?
59. *Тестовый вариант 11 (нарушены предусловия)***:**
60. *ИД:* M=15, 10, 5, 25, 20, 40, 35; Key=35.
61. *ОЖ.РЕЗ.:* Аварийное завершение: Массив не упорядочен.
62. Анализ граничных значений, если он применен правильно, является одним из наиболее полезных методов проектирования тестов. Однако он часто оказывается неэффективным из-за того, что внешне выглядит простым. Граничные условия могут быть едва уловимы и, следовательно, определение их связано с большими трудностями.
63.

- ***Порядок выполнения работы***

1. Спроектировать тесты по принципу «черного ящика» для программы, разработанной в лабораторной работе № 4. Использовать схемы алгоритмов, разработанные и уточненные в лабораторных работах № 2, 3.
2. Выбрать несколько алгоритмов для тестирования и обо значить буквами или цифрами ветви этих алгоритмов.
3. Выписать пути алгоритма, которые должны быть проверены тестами для выбранного метода тестирования.
4. Записать тесты, которые позволят пройти по путям алгоритма.
5. Протестировать разработанную вами программу. Результаты оформить в виде таблиц (см. табл. Л5.1— Л5.4).

6. Проверить все виды тестов и сделать выводы об их эффективности.
7. Оформить отчет по лабораторной работе.
8. Сдать и защитить работу.

+

- ***Защита отчета по лабораторной работе***
- Отчет по лабораторной работе должен состоять из:

1. Постановки задачи.
2. Блок-схемы программ.
3. Тестов.
4. Таблиц тестирования программы.
5. Выводов по результатам тестирования (не забывайте, что целью тестирования является обнаружение ошибок в программе).

Практическая работа №3 Модульное тестирование

Цель работы

Изучить технологии модульного тестирования. Получить практические навыки по работе с Unit Testing Framework от Microsoft.

6.2 Краткие теоретические сведения

Модульное тестирование, или unit-тестирование (англ. unit testing) процесс в программировании, позволяющий проверить на корректность отдельные модули исходного кода программы.

Идея состоит в том, чтобы писать тесты для каждой нетривиальной функции или метода. Это позволяет достаточно быстро проверить, не привело ли очередное изменение кода к регрессии, то есть к появлению ошибок в уже оттестированных местах программы, а также облегчает обнаружение и устранение таких ошибок. Цель модульного тестирования — изолировать отдельные части программы и показать, что по отдельности эти части работоспособны.

Как и любая технология тестирования, модульное тестирование не позволяет отловить все ошибки программы. В самом деле, это следует из практической невозможности трассировки всех возможных путей выполнения программы, за исключением простейших случаев. Кроме того, происходит тестирование каждого из модулей по отдельности. Это означает, что ошибки интеграции, системного уровня, функций, исполняемых в нескольких модулях не будут определены. Кроме того, данная технология бесполезна для проведения тестов на производительность. Таким образом, модульное тестирование более эффективно при использовании в сочетании с другими методиками тестирования.

Для получения выгоды от модульного тестирования требуется строго следовать технологии тестирования во всём процессе разработки программного обеспечения. Нужно хранить не только записи обо всех проведённых тестах, но и обо всех изменениях исходного кода во всех модулях. С этой целью следует использовать систему контроля версий ПО. Таким образом, если более поздняя версия ПО не проходит тест, который был успешно пройден ранее, будет несложным сверить исходный код вариантов и устранить ошибку. Также необходимо убедиться в неизменном отслеживании и анализе неудачных тестов. Игнорирование этого требования приведёт к лавинообразному увеличению неудачных тестовых результатов.

На данный момент наиболее распространенными инструментами модульного тестирования платформы .NET являются библиотека NUnit и Microsoft Unit Testing Framework.

Visual Studio Unit Testing Framework – инструмент модульного тестирования для платформы .NET, встроенный в среду разработки Visual Studio 2005 и выше. Чтобы определить, что класс является тестирующим, необходимо пометить его атрибутом [TestClass]. Если класс помечен этим атрибутом, то он может содержать в себе тестовые методы. Обычно тестирующий класс называют так же, как и тестируемый, только с префиксом Test. В тестирующем классе могут содержаться тестирующие методы и обычно для всех методов тестируемого класса, которые возвращают значение, создается отдельный тестирующий метод. Тестирующий метод обычно называют, так же как и тестируемый, только с префиксом Test.

Кроме тестирующих методов в тестирующем классе могут быть методы инициализации и очистки. Метод инициализации помечается атрибутом [TestInitialize] и позволяет инициализировать необходимые переменные перед выполнением метода-теста. Метод очистки помечается атрибутом [TestCleanup] и позволяет очистить результаты выполнения теста, например, очистить файл, удалить лишние записи с базы данных, присвоить переменным значения по умолчанию.

+Кроме методов инициализации и очистки на уровне теста, в тестирующем классе могут присутствовать методы инициализации и очистки уровня класса. Эти методы

вызываются один раз. Методы инициализации уровня класса вызывается один раз перед вызовом первого теста, а метод очистки уровня класса вызывается после выполнения последнего теста. Метод инициализации уровня класса помечается атрибутом [ClassInitialize], а метод очистки уровня класса помечается атрибутом [ClassCleanup].

Выполнение работы

Текст TestGenericDAO

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using NHibernate;
using NHibernate.Criterion;
using System.Collections.Generic;
using FluentNHibernate;
using FluentNHibernate.Cfg;
using FluentNHibernate.Cfg.Db;
using FluentNHibernate.Automapping;
using NHibernate.Tool.hbm2ddl;
using System.Reflection;
using FluentNHibernate.Mapping;
using NHibernate.Cfg;
using lab6.mapping;
using lab6.dao;
using lab6.domain;
namespace lab6
{
[TestClass()]
public abstract class TestGenericDAO<T> where T : EntityBase
{
protected static ISessionFactory factory;
protected static ISession session;
protected DaoFactory daoFactory;
protected TestContext testContextInstance;
/** DAO that will be tested */
protected IGenericDAO<T> dao = null;
/** First entity that will be used in tests */
protected T entity1 = null;
/** Second entity that will be used in tests */
protected T entity2 = null;
/** Third entity that will be used in tests */
protected T entity3 = null;
public TestGenericDAO()
{
session = openSession("localhost", 5432, "market",
"postgres", "06031992");
}
public TestContext TestContext
{
get
{
return testContextInstance;
}
set
{
```

```
testContextInstance = value;
}
}
/*Getting dao this test case works with*/
public IGenericDAO<T> getDAO()
{
return dao;
}
/*Setting dao this test case will work with*/
public void setDAO(IGenericDAO<T> dao)
{
this.dao = dao;
}
[ClassCleanup]
public static void ClassCleanup()
{
session.Close();
}
[TestInitialize]
public void TestInitialize()
{
Assert.IsNotNull(dao,
"Please, provide IGenericDAO implementation in constructor");
createEntities();
Assert.IsNotNull(entity1, "Please, create object for entity1");
Assert.IsNotNull(entity2, "Please, create object for entity2");
Assert.IsNotNull(entity3, "Please, create object for entity3");
checkAllPropertiesDiffer(entity1, entity2);
checkAllPropertiesDiffer(entity1, entity3);
checkAllPropertiesDiffer(entity2, entity3);
saveEntitiesGeneric();
}
[TestCleanup]
public void TestCleanup()
{
try
{
if ((entity1 = dao.GetById(entity1.Id)) != null)
dao.Delete(entity1);
}
catch (Exception)
{
Assert.Fail("Problem in cleanup method");
}
try
{
if ((entity2 = dao.GetById(entity2.Id)) != null)
dao.Delete(entity2);
}
catch (Exception)
{
Assert.Fail("Problem in cleanup method");
```

```
}
try
{
if ((entity3 = dao.GetById(entity3.Id)) != null)
dao.Delete(entity3);
}
catch (Exception)
{
Assert.Fail("Problem in cleanup method");
}
entity1 = null;
entity2 = null;
entity3 = null;
}
[TestMethod]
public void TestGetByIdGeneric()
{
T foundObject = null;
// Should not find with inexistent id
try
{
long id = DateTime.Now.ToFileTime();
foundObject = dao.GetById(id);
Assert.IsNull(foundObject, "Should return null if id is inexistent");
}
catch (Exception)
{
Assert.Fail("Should return null if object not found");
}
// Getting all three entities
getEntityGeneric(entity1.Id, entity1);
getEntityGeneric(entity2.Id, entity2);
getEntityGeneric(entity3.Id, entity3);
}
[TestMethod]
public void TestGetAllGeneric()
{
List<T> list = getListOfAllEntities();
Assert.IsTrue(list.Contains(entity1),
"After dao method GetAll list should contain entity1");
Assert.IsTrue(list.Contains(entity2),
"After dao method GetAll list should contain entity2");
Assert.IsTrue(list.Contains(entity3),
"After dao method GetAll list should contain entity3");
}
[TestMethod]
public void TestDeleteGeneric()
{
try
{
dao.Delete((T)null);
Assert.Fail("Should not delete entity will null id");
```

```
}
catch (Exception)
{
}
// Deleting second entity
try
{
dao.Delete(entity2);
}
catch (Exception)
{
Assert.Fail("Deletion should be successful of entity2");
}
// Checking if other two entities can be still found
getEntityGeneric(entity1.Id, entity1);
getEntityGeneric(entity3.Id, entity3);
// Checking if entity2 can not be found
try
{
T foundEntity = null;
foundEntity = dao.GetById(entity2.Id);
Assert.IsNull(foundEntity,
"After deletion entity should not be found with id " + entity2.Id);
}
catch (Exception)
{
Assert.Fail("Should return null if finding the deleted entity");
}
// Checking if other two entities can still be found in getAll list
List<T> list = getListOfAllEntities();
Assert.IsTrue(list.Contains(entity1),
"After dao method GetAll list should contain entity1");
Assert.IsTrue(list.Contains(entity3),
"After dao method GetAll list should contain entity3");
}
protected abstract void createEntities();
protected abstract void checkAllPropertiesDiffer(T entityToCheck1,
T entityToCheck2);
protected abstract void checkAllPropertiesEqual(T entityToCheck1,
T entityToCheck2);
protected void saveEntitiesGeneric()
{
T savedObject = null;
try
{
dao.SaveOrUpdate(entity1);
savedObject = getPersistentObject(entity1);
Assert.IsNotNull(savedObject,
"DAO method saveOrUpdate should return entity if successfull");
checkAllPropertiesEqual(savedObject, entity1);
entity1 = savedObject;
}
```

```
        catch (Exception)
        {
        Assert.Fail("Fail to save entity1");
        }
        try
        {
        dao.SaveOrUpdate(entity2);
        savedObject = getPersistentObject(entity2);
        Assert.IsNotNull(savedObject,
        "DAO method saveOrUpdate should return entity if successfull");
        checkAllPropertiesEqual(savedObject, entity2);
        entity2 = savedObject;
        }
        catch (Exception)
        {
        Assert.Fail("Fail to save entity2");
        }
        try
        {
        dao.SaveOrUpdate(entity3);
        savedObject = getPersistentObject(entity3);
        Assert.IsNotNull(savedObject,
        "DAO method saveOrUpdate should return entity if successfull");
        checkAllPropertiesEqual(savedObject, entity3);
        }
        catch (Exception)
        {
        Assert.Fail("Fail to save entity3");
        }
        }
        protected T getPersistentObject(T nonPersistentObject)
        {
        ICriteria                                criteria                                =
session.CreateCriteria(typeof(T)).Add(Example.Create(nonPersistentObject));
        IList<T> list = criteria.List<T>();
        Assert.IsTrue(list.Count >= 1,
        "Count of grups must be equal or more than 1");
        return list[0];
        }
        protected void getEntityGeneric(long id, T entity)
        {
        T foundEntity = null;
        try
        {
        foundEntity = dao.GetById(id);
        Assert.IsNotNull(foundEntity,
        "Service method getEntity should return entity if successfull");
        checkAllPropertiesEqual(foundEntity, entity);
        }
        catch (Exception)
        {
        Assert.Fail("Failed to get entity with id " + id);
```

```
}
}
protected List<T> getListOfAllEntities()
{
List<T> list = null;
// Should get not null and not empty list
try
{
list = dao.GetAll();
}
catch (Exception)
{
Assert.Fail(
"Should be able to get all entities that were added before");
}
Assert.IsNotNull(list,
"DAO method GetAll should return list of entities if successfull");
Assert.IsFalse(list.Count == 0,
"DAO method should return not empty list if successfull");
return list;
}
//Метод открытия сессии
public static ISession openSession(String host, int port,
String database, String user, String passwd)
{
ISession session = null;
if (factory == null)
{
FluentConfiguration configuration = Fluently.Configure()
.Database(PostgreSQLConfiguration
.PostgreSQL82.ConnectionString(c => c
.Host(host)
.Port(port)
.Database(database)
.Username(user)
.Password(passwd)))
.Mappings(m => m.FluentMappings.Add<TovarsMap>().Add<ProdMap>())
.ExposeConfiguration(BuildSchema);
factory = configuration.BuildSessionFactory();
}
//Открытие сессии
session = factory.OpenSession();
return session;
}
//Метод для автоматического создания таблиц в базе данных
private static void BuildSchema(Configuration config)
{
new SchemaExport(config).Create(false, true);
}
}
}
```

Текст TestProdDAO

```
using System;
using System.Collections.Generic;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using lab6.domain;
using lab6.dao;
namespace lab6
{
[TestClass]
public class TestProdDAO : TestGenericDAO<Prod>
{
protected IProdDAO prodDAO = null;
protected Tovars tovar1 = null;
protected Tovars tovar2 = null;
protected Tovars tovar3 = null;
public TestProdDAO()
: base()
{
DaoFactory daoFactory = new NHibernateDAOFactory(session);
prodDAO = daoFactory.getProdDAO();
setDAO(prodDAO);
}
protected override void createEntities()
{
entity1 = new Prod();
entity1.NameProd = "1";
entity1.Surname = "1";
entity1.God = 1992;
entity2 = new Prod();
entity2.NameProd = "2";
entity2.Surname = "2";
entity2.God = 1993;
entity3 = new Prod();
entity3.NameProd = "3";
entity3.Surname = "3";
entity3.God = 1994;
}
protected override void checkAllPropertiesDiffer(Prod entityToCheck1,
Prod entityToCheck2)
{
Assert.AreNotEqual(entityToCheck1.NameProd, entityToCheck2.NameProd, "Values must be different");
Assert.AreNotEqual(entityToCheck1.Surname,entityToCheck2.Surname, "Values must be different");
Assert.AreNotEqual(entityToCheck1.God,entityToCheck2.God, "Values must be different");
}
protected override void checkAllPropertiesEqual(Prod entityToCheck1,
Prod entityToCheck2)
{
Assert.AreEqual(entityToCheck1.NameProd, entityToCheck2.NameProd,
"Values must be equal");
Assert.AreEqual(entityToCheck1.Surname, entityToCheck2.Surname,
```

```
"Values must be equal");
Assert.AreEqual(entityToCheck1.God, entityToCheck2.God,
"Values must be equal");
}
[TestMethod]
public void TestGetByIdProd()
{
base.TestGetByIdGeneric();
}
[TestMethod]
public void TestGetAllProd()
{
base.TestGetAllGeneric();
}
[TestMethod]
public void TestDeleteProd()
{
base.TestDeleteGeneric();
}
// [TestMethod]
// public void TestGetProdByName()
// {
// Prod prod1 = prodDAO.getProdByName(entity1.NameProd);
// Assert.IsNotNull(prod1,
// "Service method getGroupByName should return group if successfull");
// Prod prod2 = prodDAO.getProdByName(entity2.NameProd);
// Assert.IsNotNull(prod2,
// "Service method getGroupByName should return group if successfull");
// Prod prod3 = prodDAO.getProdByName(entity3.NameProd);
// Assert.IsNotNull(prod3,
// "Service method getGroupByName should return group if successfull");
// checkAllPropertiesEqual(prod1, entity1);
// checkAllPropertiesEqual(prod2, entity2);
// checkAllPropertiesEqual(prod3, entity3);
// }
[TestMethod]
public void TestGetAllTovarsOfProd()
{
createEntitiesForTovars();
Assert.IsNotNull(tovar1, "Please, create object for student1");
Assert.IsNotNull(tovar2, "Please, create object for student2");
Assert.IsNotNull(tovar3, "Please, create object for student3");
entity1.TovarsList.Add(tovar1);
tovar1.Prod = entity1;
entity1.TovarsList.Add(tovar2);
tovar2.Prod = entity1;
entity1.TovarsList.Add(tovar3);
tovar3.Prod = entity1;
Prod savedObject = null;
try
{
dao.SaveOrUpdate(entity1);
```

```
savedObject = getPersistentObject(entity1);
Assert.IsNotNull(savedObject,
"DAO method saveOrUpdate should return entity if successfull");
checkAllPropertiesEqual(savedObject, entity1);
entity1 = savedObject;
}
catch (Exception)
{
Assert.Fail("Fail to save entity1");
}
IList<Tovars> tovarsList =
prodDAO.getAllTovarsOfProd(entity1.NameProd);
Assert.IsNotNull(tovarsList, "List can't be null");
Assert.IsTrue(tovarsList.Count == 3,
"Count of students in the list must be 3");
checkAllPropertiesEqualForTovars(tovarsList[0], tovar1);
checkAllPropertiesEqualForTovars(tovarsList[1], tovar2);
checkAllPropertiesEqualForTovars(tovarsList[2], tovar3);
}
//[TestMethod]
//public void TestDelProdByName()
//{
// try
// {
// prodDAO.delProdByName(entity2.NameProd);
// }
// catch (Exception)
// {
// Assert.Fail("Deletion should be successful of entity2");
// }
// Checking if other two entities can be still found
// getEntityGeneric(entity1.Id, entity1);
// getEntityGeneric(entity3.Id, entity3);
// Checking if entity2 can not be found
// try
// {
// Prod foundProd = null;
// foundProd = dao.GetById(entity2.Id);
// Assert.IsNull(foundProd,
// "After deletion entity should not be found with groupName " +
// entity2.NameProd);
// }
// catch (Exception)
// {
// Assert.Fail("Should return null if finding the deleted entity");
// }
// Checking if other two entities can still be found in getAll list
// List<Prod> list = getListOfAllEntities();
// Assert.IsTrue(list.Contains(entity1),
// "After dao method GetAll list should contain entity1");
// Assert.IsTrue(list.Contains(entity3),
// "After dao method GetAll list should contain entity3");
```

```
// }
protected void createEntitiesForTovars()
{
tovar1 = new Tovars();
tovar1.Name = "C";
tovar1.Price = 10;
tovar1.Ves = 100;
tovar2 = new Tovars();
tovar2.Name = "B";
tovar2.Price = 20;
tovar2.Ves = 200;
tovar3 = new Tovars();
tovar3.Name = "D";
tovar3.Price = 30;
tovar3.Ves = 300;
}
protected void checkAllPropertiesEqualForTovars(Tovars entityToCheck1,
Tovars entityToCheck2)
{
Assert.AreEqual(entityToCheck1.Name, entityToCheck2.Name,
"Values must be equal");
Assert.AreEqual(entityToCheck1.Price, entityToCheck2.Price,
"Values must be equal");
Assert.AreEqual(entityToCheck1.Ves, entityToCheck2.Ves,
"Values must be equal");
}
}
}
```

Текст TestTovarsDAO

```
using System;
using System.Collections.Generic;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using lab6.domain;
using lab6.dao;
using NHibernate.Criterion;
using NHibernate;
namespace lab6
{
[TestClass]
public class TestTovarsDAO : TestGenericDAO<Tovars>
{
protected ITovarsDao tovarsDAO = null;
protected IProdDAO prodDAO = null;
protected Prod prod = null;
public TestTovarsDAO()
: base()
{
DaoFactory daoFactory = new NHibernateDAOFactory(session);
tovarsDAO = daoFactory.getTovarsDAO();
prodDAO = daoFactory.getProdDAO();
setDAO(tovarsDAO);
}
```

```
protected override void createEntities()
{
entity1 = new Tovars();
entity1.Name = "C";
entity1.Price = 10;
entity1.Ves = 100;
entity2 = new Tovars();
entity2.Name = "B";
entity2.Price = 20;
entity2.Ves = 200;
entity3 = new Tovars();
entity3.Name = "D";
entity3.Price = 30;
entity3.Ves = 300;
}
protected override void checkAllPropertiesDiffer(Tovars entityToCheck1,
Tovars entityToCheck2)
{
Assert.AreNotEqual(entityToCheck1.Name, entityToCheck2.Name,
"Values must be different");
Assert.AreNotEqual(entityToCheck1.Price, entityToCheck2.Price,
"Values must be different");
Assert.AreNotEqual(entityToCheck1.Ves, entityToCheck2.Ves,
"Values must be different");
}
protected override void checkAllPropertiesEqual(Tovars entityToCheck1,
Tovars entityToCheck2)
{
Assert.AreEqual(entityToCheck1.Name, entityToCheck2.Name,
"Values must be equal");
Assert.AreEqual(entityToCheck1.Price, entityToCheck2.Price,
"Values must be equal");
Assert.AreEqual(entityToCheck1.Ves, entityToCheck2.Ves,
"Values must be equal");
}
[TestMethod]
public void TestGetByIdTovars()
{
base.TestGetByIdGeneric();
}
[TestMethod]
public void TestGetAllTovars()
{
base.TestGetAllGeneric();
}
[TestMethod]
public void TestDeleteTovars()
{
base.TestDeleteGeneric();
}
[TestMethod]
public void TestGetTovarByProdName()
```

```
{
Prod prod = new Prod();
prod.NameProd = "1";
prod.Surname = "1";
prod.God = 1992;
prod.TovarsList.Add(entity1);
entity1.Prod = prod;
prod.TovarsList.Add(entity2);
entity2.Prod = prod;
prod.TovarsList.Add(entity3);
entity3.Prod = prod;
Prod savedProd = null;
try
{
prodDAO.SaveOrUpdate(prod);
savedProd = getPersistentProd(prod);
Assert.IsNotNull(savedProd,
"DAO method saveOrUpdate should return group if successfull");
checkAllPropertiesEqualProd(savedProd, prod);
prod = savedProd;
}
catch (Exception)
{
Assert.Fail("Fail to save group");
}
getTovarByProdName(entity1, prod.NameProd,
entity1.Name);
getTovarByProdName(entity2, prod.NameProd,
entity2.Name);
getTovarByProdName(entity3, prod.NameProd,
entity3.Name);
prod.TovarsList.Remove(entity1);
prod.TovarsList.Remove(entity2);
prod.TovarsList.Remove(entity3);
entity1.Prod = null;
entity2.Prod = null;
entity3.Prod = null;
prodDAO.Delete(prod);
}
protected void getTovarByProdName(Tovars tovar, string nameProd, string name)
{
Tovars foundTovars = null;
try
{
foundTovars = tovarsDAO.getTovarByProdName(
nameProd,name);
Assert.IsNotNull(tovarsDAO,
"Service method should return student if successfull");
checkAllPropertiesEqual(foundTovars, tovar);
}
catch (Exception)
{
```

```
Assert.Fail("Failed to get student with nameProd " +
nameProd + " name " + name);
}
}
protected Prod getPersistentProd(Prod nonPersistentProd)
{
ICriteria criteria = session.CreateCriteria(typeof(Prod))
.Add(Example.Create(nonPersistentProd));
IList<Prod> list = criteria.List<Prod>();
Assert.IsTrue(list.Count >= 1,
"Count of grups must be equal or more than 1");
return list[0];
}
protected void checkAllPropertiesEqualProd(Prod entityToCheck1,
Prod entityToCheck2)
{
Assert.AreEqual(entityToCheck1.NameProd, entityToCheck2.NameProd,
"Values must be equal");
Assert.AreEqual(entityToCheck1.Surname, entityToCheck2.Surname,
"Values must be equal");
Assert.AreEqual(entityToCheck1.God, entityToCheck2.God,
+"Values must be equal");
}
}
}
```

Выводы

В ходе выполнения данной лабораторной работы были изучены технологии модульного тестирования. Были получены практические навыки по работе с Unit Testing Framework от Microsoft.

Модульное тестирование, или unit-тестирование (англ. unit testing) процесс в программировании, позволяющий проверить на корректность отдельные модули исходного кода программы.

Практическая работа №4 Интеграционное тестирование

Цель работы

Овладение навыками интеграционного тестирования.

Общие сведения

Интеграционное тестирование называют еще тестированием архитектуры системы. С одной стороны, это название обусловлено тем, что интеграционные тесты включают в себя проверки всех возможных видов взаимодействий между программными модулями и элементами, которые определяются в архитектуре системы - таким образом, интеграционные тесты проверяют полноту взаимодействий в тестируемой реализации системы. С другой стороны, результаты выполнения интеграционных тестов - один из основных источников информации для процесса улучшения и уточнения архитектуры системы, межмодульных и межкомпонентных интерфейсов. Т.е., с этой точки зрения, интеграционные тесты проверяют корректность взаимодействия компонент системы.

В результате проведения интеграционного тестирования и устранения всех выявленных дефектов получается согласованная и целостная архитектура программной системы, т.е. можно считать, что интеграционное тестирование - это тестирование архитектуры и низкоуровневых функциональных требований. Интеграционное тестирование, как правило, представляет собой итеративный процесс, при котором проверяется совокупность модулей, возрастающая от итерации к итерации. В интеграционном тестирование выделяют три методы выполнения: восходящее тестирование; монолитное тестирование; нисходящее тестирование.

Задание

Согласно варианту провести один из методов интеграционного тестирования.

Практическая работа №5 Оформление документации на программные средства с использованием инструментальных средств.

Цель работы

Овладение навыками документирования результатов тестирования.

Общие сведения

Каждый дефект, обнаруженный в процессе тестирования, должен быть задокументирован и отслежен. При обнаружении нового дефекта его заносят в базу дефектов. При занесении нового дефекта рекомендуется указывать, как минимум, следующую информацию:

1) Наименование подсистемы, в которой обнаружен дефект.
2) Версия продукта (номер build), на котором дефект был найден.
3) Описание дефекта.
4) Описание процедуры (шагов, необходимых для воспроизведения дефекта).
5) Номер теста, на котором дефект был обнаружен.
6) Уровень дефекта, то есть степень его серьезности с точки зрения критериев качества продукта или заказчика.

Тестовый отчет обновляется после каждого цикла тестирования и должен содержать следующую информацию для каждого цикла:

Задание

Задокументировать результаты тестирования.