

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ**

**ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ**

«ДОНСКОЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

**(ДГТУ)**

**Методические указания и задания для контрольной работы**

**по дисциплине «Системная инженерия»**

Предназначено для магистров 2˗го курса заочной формы обучения по направлению 09.04.02 Информационные системы и технологии

г. Ростов-на-Дону

2023 год

Кафедра Вычислительные системы и информационная безопасность

Доцент Анисимова Г.Б.

Методические указания содержат контрольные работы по курсу «Системная инженерия» для студентов направления 09.04.02 заочной формы обучения и рекомендации по их выполнению.

Содержание

[Требования к оформлению 4](#_Toc144160488)

[Введение 5](#_Toc144160489)

[Часть 1 6](#_Toc144160490)

[Проектирование БД 6](#_Toc144160491)

[Лабораторая работа 1. 6](#_Toc144160492)

[Лабораторная работа 2. 8](#_Toc144160493)

[Лабораторная работа 3. 10](#_Toc144160494)

[Часть 2 13](#_Toc144160495)

[Тестирование 13](#_Toc144160496)

[Лабораторная №1. 13](#_Toc144160497)

[Лабораторная №2. 21](#_Toc144160498)

[Лабораторная №3. 26](#_Toc144160499)

[Контрольная работа №1 38](#_Toc144160500)

[Контрольная работа №2 39](#_Toc144160501)

[Контрольная работа №3 40](#_Toc144160502)

[Контрольная работа №4 41](#_Toc144160503)

[Список литературы 42](#_Toc144160504)

# Требования к оформлению

В отчет по контрольной работе должны быть включены следующие пункты:

• титульный лист;

• содержание;

• цель работы;

• краткие теоретические сведения;

• подробное описание этапов реализации;

• анализ результатов работы;

• выводы.

# Введение

Основное назначение представленного цикла лабораторных и контрольных работ – знакомство с работами 2 и 3 этапов жизненного цикла АИС (ЖЦ). Это этапы Проектирования, Реализации и тестирования анализа предметной области и формулирование требований к АИС.

Знакомство выполняется на примере предприятия, связанного с темой НИР.

Применяется на 2 этапе методология структурного подхода с использованием стандартов IDEF0 и DFD, или объектно-ориентированного на языке UML.

На этапе тестирования представлены различные методологии тестирования АИС.

# Часть 1

# Проектирование БД

# Лабораторая работа 1.

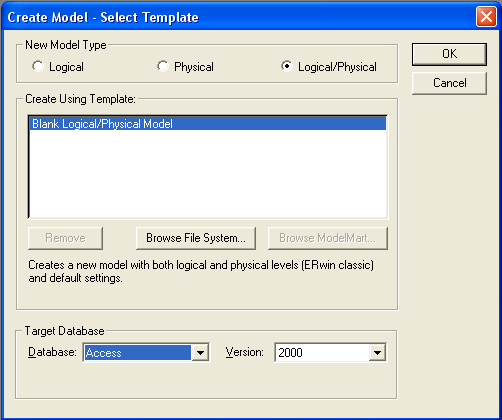
1. Запустите программу ERwin 4.0

2. Создайте новую модель. create a new model.

Укажите, что создается логически-физическая модель. Логически-физическая модель можно создавать для последующего создания базы данных в полностью русифицированных СУБД, например, Access (названия атрибутов и таблиц могут быть русскими). В остальных случаях логическую и физическую модель разделяют. В логической названия сущностей и атрибутов – русские. А в физической используется латиница.

Logical/Physical,

Target Database - Access



3. Автоматически создается модель. На панели инструментов выберите логическую модель. Logical.

4. При помощи кнопки  (entity) на панели инструментов создайте сущность "Сотрудник". Задайте для сущности "Сотрудник" следующие атрибуты:

Табельный номер

Фамилия

Имя

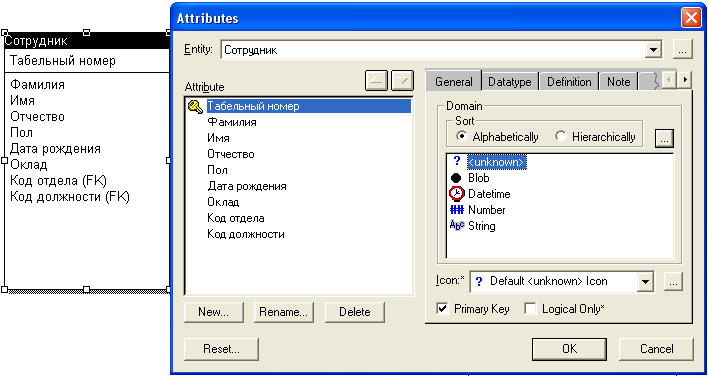
Отчество

Пол

Дата рождения

Оклад

Определите атрибут табельный номер как ключевой атрибут (Primary Key)

.

5. Создайте сущности "Отдел", "Должность", "Дети сотрудников".

Создайте для сущности "Отдел " следующие атрибуты:

Код отдела

Название отдела

Создайте для сущности "Должность" следующие атрибуты:

Код должности

Название должности

Создайте для сущности "Дети сотрудников " следующие атрибуты:

ID номер

Фамилия

Имя

Отчество

Пол

Дата рождения

Определите атрибуты "Код отдела", "Код должности", "ID номер" как ключевые атрибуты

6. Установите связи для сущностей:

"Отдел" и "Сотрудник", "Должность" "Сотрудник" - Неидентифицирующие связи;

"Сотрудник" и "Дети сотрудников" - идентифицирующая связь



В результате у сущности "Сотрудник" появились атрибуты "Код отдела" "Код должности", помеченные как Foreign Key (FK).А у сущности"Дети сотрудников" ключевой атрибут "Табельный номер" так же помеченный как(FK).

# Лабораторная работа 2.

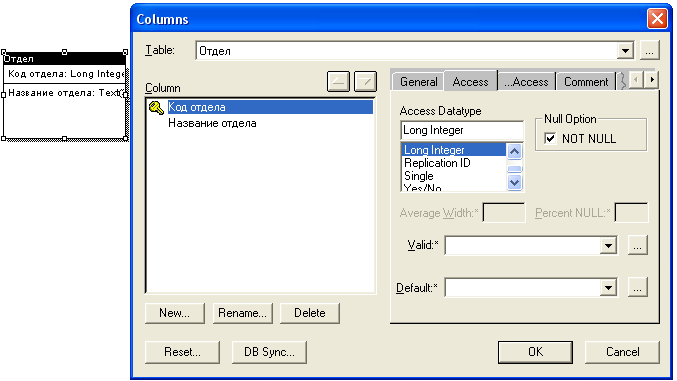
Создание физической модели с помощью ERwin

1. На панели инструментов выберите физическую модель. Physical.

Для сущности "Отдел " задайте следующие типы атрибутов:

Код отдела – Long integer

Название отдела – Text(30)



Для сущности " Должность " задайте следующие типы атрибутов:

Код должности – Long integer

Название должности – Text(30)

Для сущности " Дети сотрудников " задайте следующие типы атрибутов:

ID номер– Text(1)

Фамилия– Text(20)

Имя– Text(12)

Отчество– Text(15)

Пол– Text(1)

Дата рождения - Date/Time

Для сущности " Сотрудник " задайте следующие типы атрибутов:

Табельный номер– Text(4)

Фамилия– Text(20)

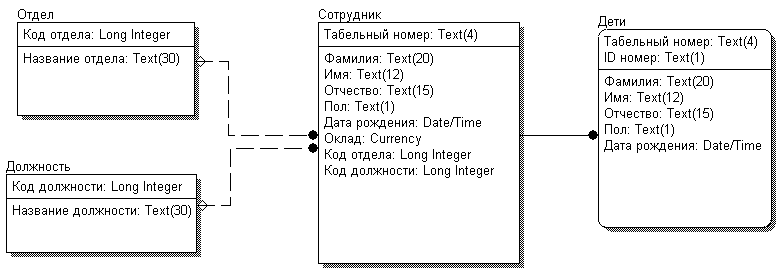
Имя– Text(12)

Отчество– Text(15)

Пол– Text(1)

Дата рождения - Date/Time

Оклад - Currency



2.

# Лабораторная работа 3.

Создание базы данных

1. Запустите программу Access.

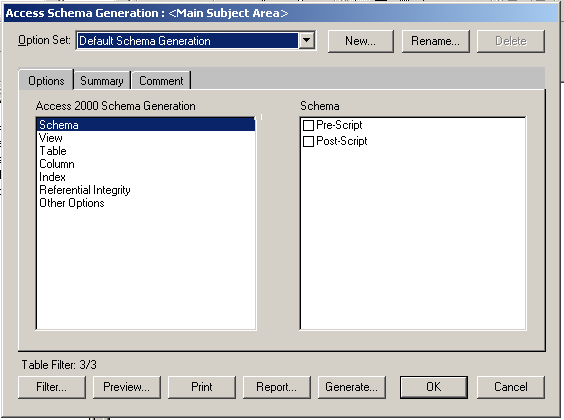
Создайте новую базу данных, и сохраните на своем диске.

Закройте программу Access.

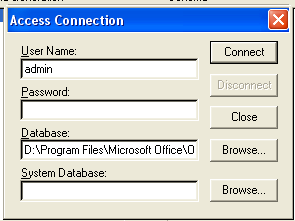
2. Запустите программу ERwin 4.0

Откройте физическую модель.

При помощи кнопки  (Forward Engineer) на панели инструментов проведите прямое проектирование. В появившемся окне выберите Generate.



В окне Access Connection в поле User Name установите имя пользователя admin. В поле Database укажите путь к созданному в Access файлу базы данных. Нажмите кнопку Connect.



Начнется процесс генерации базы данных. В появившемся окне вы можете следить за ходом процесса. По окончании процесса генерации нажмите кнопку ОК.

3. Запустите программу Access. Откройте созданную вами базу данных.

Убедитесь в том, что на основе сущностей были созданы таблицы в базе данных.

Откройте таблицы в режиме конструктора (кнопка  панели инструментов), и убедитесь в том, что

на основе атрибутов были созданы поля в таблицах;

тип данных длина поля соответствует типу данных и длине поля в ERD модели

ключевые поля соответствует ключевым полям, выбранным в ERD модели

Откройте схему данных (кнопка  панели инструментов), и убедитесь в наличии связей между таблицами, связи соответствуют связям, установленным при создании модели в Erwin

# Часть 2

# Тестирование

# Лабораторная №1.

**Тестирование методом «белого ящика»**

Цель. Разработать тесты для белого ящика, проверяющего работоспособность программного продукта.

Программа предназначена для определения асимптот функций одного аргумента у =f(х). При этом считается, что функция задана аналитически в виде отношения: , где и – полиномиальные функции вида: и . Кроме того, разрабатываемая программа должна рассчитывать таблицу значений и строить график данной функций и его асимптот на заданном отрезке по заданной формуле и менять шаг аргумента и границы отрезка.

*Порядок выполнения работы*

1. Спроектировать тесты по принципу «белого ящика» для программы.
2. Разработать блок-схему алгоритмов определения асимптот.
3. Обозначить буквами или цифрами ветви этих алгоритмов.
4. Выписать пути алгоритма, которые должны быть проверены тестами для выбранного метода тестирования.
5. Записать тесты, которые позволят пройти по путям алгоритма.
6. Протестировать разработанную вами программу. Результаты оформить в виде таблиц (см. табл. Л5.1— Л5.4 Приложения).
7. Проверить все виды тестов и сделать выводы об их эффективности.
8. Оформить отчет по лабораторной работе.
9. Сдать и защитить работу.

*Варианты*

Тестирование осуществляется для алгоритма вычисления по вариантам:

1. , .
2. , .
3. , .
4. , .
5. , .
6. , .
7. , .
8. , .
9. , .
10. , .
11. , .
12. , .
13. , .
14. , .
15. , .
16. , .
17. , .
18. , .
19. , .
20. , .
21. , .
22. , .
23. , .
24. , .
25. , .

**Выбор варианте тестирования определяется по двум последним цифрам зачетки \*\*\*XY. Правило определения варианта: для X={1,…,5} варианты с 1 по 10, для X={6,…,10} варианты с 11 по 20.**

Необходимо реализовать блок-схему определяющую правильность ввода граничных условий для функции, области возрастания и убывания

*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).

**Приложение:**

*Теоретическая часть.*

***Виды тестирования***

Тестирование программного обеспечения включает в себя целый комплекс действий, аналогичных последовательности процессов разработки программного обеспечения. В него входят [7]:

* 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. В данной лабораторной рассматривается тестирование на основе стратегии «белого ящика», основанной на анализе логики программы. При таком подходе тестирование заключается в проверке каждого пути, каждой ветви алгоритма. При этом внешняя спецификация во внимание не принимается.

Тестирование методом «белого ящика» 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:

*? (А + В+ С)/3=Д,*

то оно будет верным не для всех значений *А, В* и С (ошибка возникает в том случае, когда из двух значений *В* или *С* одно больше, а другое на столько же меньше *А).* 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.

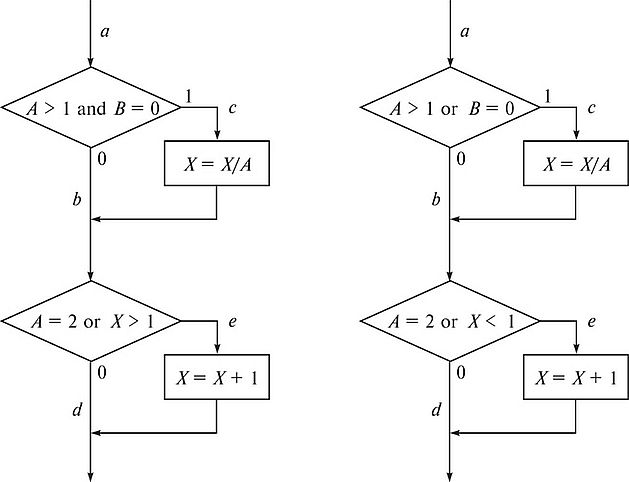
Пример.

Если для тестирования задать значения переменных *А* = 2, *В=* 0, *Х=3,* будет реализован путь *асе,* 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 *Х>* 1 на *Х<* 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 |
| *А = 2, В=* 0, *Х=3* | *Х=* 2,5 | *Х=* 2,5 | Неуспешно |



*а б*

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 *{асе, аЬd*}, либо *{асd, аЬе*}. Для этого выберем следующие исходные данные; *{А =* 3, *В=* 0, *Х=3} —* в первом случае и *{А = 2, В=* 1, *Х=* 1} — во втором. Однако путь, где *X* не меняется, будет проверен с вероятностью 50 %: если во втором условии вместо условия *Х>* 1 записано *Х<* 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 |
| /1 = 3, *В=* 0, *Х=2* | *X* = 1 | *X* = 1 | Неуспешно |
| *А = 2, В =* 1, *Х=* 1 | *Х=2* | \*  II | 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 четыре условия: *{А>* 1, 5=0}, *{А* = 2, *Х>* 1}. Следовательно, требуется достаточное число тестов, такое, чтобы реализовать ситуации, где *А* > 1, *А <* 1, 5 = 0 и й#0 в точке а и /1 = 2, *А \*2, Х>* и Т< 1 в точке *Ь.* Тесты, удовлетворяющие критерию покрытия условий (табл. Л5.3), и соответствующие им пути:

* а) *А = 2,* 5 = 0, *Х=4 асе;*
* б) *А =* 1, 5= 1, *Х=0 аЬс!.*

*Таблица Л5.3.* Результаты тестирования методом покрытия условий

|  |  |  |  |
| --- | --- | --- | --- |
| Тест | Ожидаемый | Фактический | Результат |
| результат | результат | тестирования |
| /1 = 2, *В=* 0, *Х= 4* | \*=з | *Х= 3* | Неуспешно |
| *А=* 1, *В=* 1, *Х=0* | \*=0 | *Х=* 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.

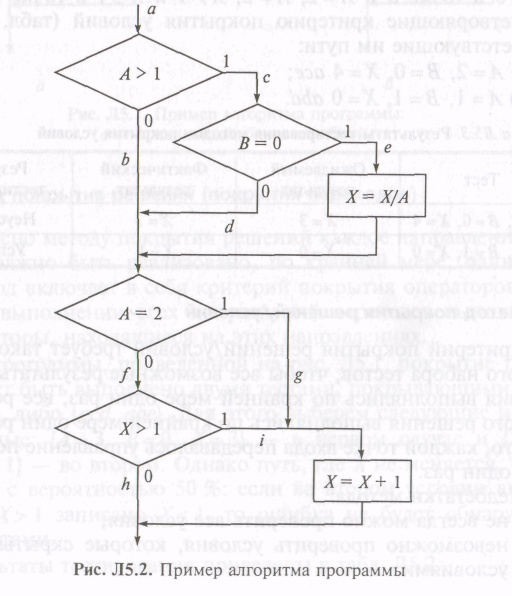
Так, в рассматриваемом примере два теста метода покрытия условий

а) *А* = 2, *В =* О, *Х= 4 асе;*

б*) А=* 1, *В=* 1, *Х=0 аbd*

отвечают и критерию покрытия решений/условий. Это является следствием того, что одни условия приведенных решении скрывают другие условия в этих решениях. Так, если условие *А >* 1 будет ложным, транслятор может не проверять условия *В=* О, поскольку при любом результате условия *В=* 0 результат решения *((А > )&(В=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 А = 2, B=0, X=4), *acdfh* (тест A = 3, B=1, X=0), *abfh* (тест A = 0, B = 0, Х= 0), *abfi* (тест А = 0, B= 0, Х= 2).



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.
2. А = 2, Х>1.
3. A > 1, B<>0.
4. А = 2, Х<1.
5. A<1, B=0.
6. А<>2, Х>1.
7. 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. Л5.4):

* *А =* 2, *В = 0, Х=4* (покрывает 1,5};
* *А = 2, В = 1, Х= 1* (покрывает 2,6};
* *А =* 0,5, *В=* 0, *Х= 2* (покрывает 3, 7};
* *А = I, В = 0, Х=* (покрывает 4, 8}.

*Таблица Л5.4.* Результаты тестирования методом комбинаторного покрытия условий

|  |  |  |  |
| --- | --- | --- | --- |
| Тест | Ожидаемый результат | Фактический результат | Результат тестирования |
| *А-2, В=0, X-* 4 | X=3 | X=з | Неуспешно |
| *А = 2, В=* 1, *Х=* 1 | *Х= 2* | X=1,5 | Успешно |
| *А =* 0,5, *В=* 0, *Х=2* | *Х=3* | *Х= 4* | Успешно |
| *А=* 1, B=0, \*= 1 | *Х=* 1 | *Х=* 1 | Неуспешно |

# Лабораторная №2.

**Тестирование методом «черного** **ящика»**

Цель. Разработать тесты для черного ящика, проверяющего работоспособность информационной системы, учитывающей клиентов и проводящей анализ их возраста и места проживания. Диалоговое окно ввода личных данных клиента: ввод ФИО, даты рождения, пола, паспортных данных, места проживания. Требование по возрасту для клиентов: 18-60 лет для женщин и 18-65 лет для мужчин.

Результатом обработки данных являются отчеты:

1. о численности клиентов в возрастных диапазонах 18-30, 30-45, 45-60 для женщин и 18-25, 25-40, 40-55, 55-65 для мужчин;
2. о распределении клиентов по районам и городам Ростовской области.

Основы создания тестов черного ящика в документе:

*9 Создание тестов черного ящика*.

Для выполнения работы определите:

1. Правила ввода и граничные условия для каждого из данных
2. Требования по срабатыванию ошибочных ситуаций
3. Классы эквивалентности для входных данных
4. Разработать тестовые наборы данных на основе техники **Pairwise testing** (техника формирования наборов тестовых данных, в которых каждое тестируемое значения каждого из проверяемых параметров хотя бы единожды сочетается с каждым тестируемым значением всех остальных проверяемых параметров)
5. Разработать тестовые случаи для выполнения тестов по следующим правилам: каждый тест кейс должен состоять из трех частей, как показано в таблице 1.
6. Провести тестирование системы для следующих примеров:

* эквивалентное разбиение;
* анализ граничных значений;
* анализ причинно-следственных связей;
* попарное тестирование;
* предположение об ошибке.

Таблица 1 - Структура Test case

|  |  |  |  |
| --- | --- | --- | --- |
| **Pre conditions** |  | Список действий, которые приводят систему к состоянию пригодному для проведения основной |  |
| проверки. Либо список условий, выполнение которых говорит о том, что система находится в пригодном для проведения основного теста состояния. |
| **Test case description** |  | Список действий, переводящих систему из одного состояния в другое, для получения результата | , |
| на основании которого можно сделать вывод о удовлетворении реализации, поставленным требованиям |
| **Post conditions** | Список действий, переводящих систему в первоначальное состояние (состояние до проведения теста - initial state) | |

**Примечание**: **Post Conditions** не является обязательной частью. Эта часть актуальна при автоматизированном тестировании, когда за один прогон можно наполнить базу данных сотней или даже тысячей некорректных документов. **Пример тест кейса** do A1, verify B1 do A2, verify B2 do A3, verify B3

В приведенном примере конечная проверка - В3. Это значит, что именно она является ключевой. Значит, A1 и А2 - это действия приводящие систему в тестопригодное состояние. А В1 и В2 - условия того, что система находится в состоянии пригодном для тестирования. Таким образом в таблице 2 показано условие тест кейса.

Таблица 2 - Условие тест кейса

|  |  |  |
| --- | --- | --- |
| **Action** | **Expected Result** | **Test Result (passed/failed/blocked)** |
| **Preconditions** |  |  |
| do A1 | verify B1 |  |
| do A2 | verify B2 |  |
| **Test Case Description** |  |  |
| do A3 | verify B3 |  |
| **Postconditions** |  |  |
|  |  |  |

PostConditions в данном примере не были описаны, но по логике вещей надо выполнить шаги, которые бы вернули систему в первоначальное состояние. (например, удалили созданную запись, или отменили бы изменения сделанные в документе)

Нужно ответить на вопрос: "Почему данное разбиение удобно использовать?"

Ответ в таблице 3: конечная проверка одна, т.е. в случае если тест провален (**test failed**) будет сразу ясно из-за чего. Т.к. если провальными окажутся проверки В1 и/или В2, то тест кейс будет заблокирован (**test blocked**), из-за того, что функцию не возможно привести в тестопригодное состояние (состояние пригодное для проведения тестирования), но это не значит, что тестируемая функция не работает.

Таблица 3 - Один из вариантов результата

|  |  |  |
| --- | --- | --- |
| **Action** | **Expected Result** | **Test Result (passed/failed/blocked)** |
| **Preconditions** |  |  |
| do A1 | verify B1 | passed |
| do A2 | verify B2 | *failed* |
| **Test Case Description** |  |  |
| do A3 | verify B3 | **blocked** |
| **Postconditions** |  |  |
|  |  |  |

7.4.3 Детализация описания тест кейсов

Уровень детализации тест кейсов должен быть таков, чтобы обеспечивать разумное соотношение времени прохождения к тестовому покрытию. Т.е. до тех пор, пока покрытие тестами определенного функционала не меняется, можно уменьшать детализацию тест кейсов. В таблице

7.5 можно увидеть пример детализации тест кейса:

Таблица 4 - Пример тест кейса 1

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Проверка отображения страницы** | | | | | | |  |  |
| **Действие** | | | | **Ожидаемый результат** | | |  | **Результат теста** |
|  | Открыть страницу "Вход в систему" | |  |  | - окно "Вход в систему" открыто; | в  ;  . | ... |  |
|  |  | * название окна - Вход в систему; - логотип компании отображается правом верхнем углу; * на форме 2 поля - Имя и Пароль * кнопка Вход доступна; * ссылка "забыл пароль" - доступна |
|  |

**Пример тест кейса 2**:

**Название**: Проверка отображения страницы

**Действие**: Открыть страницу "Вход в систему"

**Проверка**: Проверьте, что отображаемая страница соответствует странице на картинке 1 (и прилагаем изображение страницы "Вход в систему")

В примере 1 и 2 покрытие будет одинаковым, но вот время, которое потребуется для прохождения, будет разным. Второй пример будет нагляднее.

# Лабораторная №3.

**Модульное тестирование**

Цель. Разработать тесты для модульного тестирования, проверяющего работоспособность программного продукта. Модульное тестирование осуществляется на платформе модульных тестов Microsoft для управляемого кода.

Программа предназначена для определения асимптот функций одного аргумента у =f(х). При этом считается, что функция задана аналитически в виде отношения: , где и – полиномиальные функции вида: и . Кроме того, разрабатываемая программа должна рассчитывать таблицу значений и строить график данной функций и его асимптот на заданном отрезке по заданной формуле и менять шаг аргумента и границы отрезка.

*Порядок выполнения работы*

1. Создайте проект для задачи на основе шаблона проекта **Консольное приложение (.NET Core)** на C#.
2. Создайте классы необходимые для определения асимптот функции в соответствии с блок-схемами алгоритмов, разработанных в 1-ой лабораторной работе
3. Добавьте к проекту задачи проект модульного теста.
4. Добавьте в тестовый проект классы для проверки функций классов программы определения асимптот функции.
5. Разработайте для каждого тестового класса методы проверки работоспособности тестируемой функции.
6. Запустите тестовый проект для проверки работоспособности программы.
7. Исправьте допущенные ошибки.
8. Запустите тест повторно.
9. Сделать выводы об эффективности тестирования.
10. Оформить отчет по лабораторной работе.
11. Сдать и защитить работу.

*Варианты*

Тестирование осуществляется по вариантам лабораторной 1.

*Защита отчета по лабораторной работе*

Отчет по лабораторной работе должен состоять из:

1. Постановки задачи.
2. Листинг задачи.
3. Листинги классов для тестирования.
4. Отчет по запуску тестирования.
5. Покажите обнаруженные ошибки
6. Приведите исправленный код
7. Проведите повторное тестирование
8. Выводов по результатам тестирования (не забывайте, что целью тестирования является обнаружение ошибок в программе).

**Приложение:**

**Пошаговое руководство. Создание и запуск модульных тестов для управляемого кода**

В руководстве производится создание проекта C#, находящегося в стадии разработки, создание тестов для проверки его кода, запуск тестов и изучение результатов. После этого производится изменение кода проекта и повторный запуск тестов.

**Создайте проект для тестирования**

1. Запустите Visual Studio.
2. На начальном экране выберите **Создать проект**.
3. Найдите и выберите шаблон проекта **Консольное приложение (.NET Core)** на C# и нажмите кнопку **Далее**.
4. Присвойте проекту имя **Bank** и нажмите кнопку **Создать**.

Будет создан проект Bank. Он отобразится в **обозревателе решений**, а его файл *Program.cs* откроется в редакторе кода.

1. Замените содержимое файла *Program.cs* следующими кодом на C#, который определяет класс *BankAccount*:

**Листинг 1**

using System;

namespace BankAccountNS

{

/// <summary>

/// Bank account demo class.

/// </summary>

public class BankAccount

{

private readonly string m\_customerName;

private double m\_balance;

private BankAccount() { }

public BankAccount(string customerName, double balance)

{

m\_customerName = customerName;

m\_balance = balance;

}

public string CustomerName

{

get { return m\_customerName; }

}

public double Balance

{

get { return m\_balance; }

}

public void Debit(double amount)

{

if (amount > m\_balance)

{

throw new ArgumentOutOfRangeException("amount");

}

if (amount < 0)

{

throw new ArgumentOutOfRangeException("amount");

}

m\_balance += amount; // intentionally incorrect code

}

public void Credit(double amount)

{

if (amount < 0)

{

throw new ArgumentOutOfRangeException("amount");

}

m\_balance += amount;

}

public static void Main()

{

BankAccount ba = new BankAccount("Mr. Bryan Walton", 11.99);

ba.Credit(5.77);

ba.Debit(11.22);

Console.WriteLine("Current balance is ${0}", ba.Balance);

}

}

}

1. Переименуйте файл в *BankAccount.cs*, щелкнув его правой кнопкой мыши и выбрав команду **Переименовать** в **обозревателе решений**.
2. В меню **Сборка** выберите **Собрать решение**.

Теперь у вас есть проект с методами, которые можно протестировать. В этой статье тестирование проводится на примере метода *Debit*. Метод *Debit* вызывается, когда денежные средства снимаются со счета.

**Создание проекта модульного теста**

1. В меню **Файл** выберите **Добавить** > **Создать проект**.

**Совет**

В **обозревателе решений** щелкните решение правой кнопкой мыши и выберите пункты **Добавить** > **Создать проект**.

1. Найдите и выберите шаблон проекта **Тестовый проект MSTest (.NET Core)** на C# и нажмите кнопку **Далее**.
2. Назовите проект **BankTests**.
3. Нажмите кнопку **Создать**.

Проект **BankTests** добавляется в решение **Банк**.

1. В проекте **BankTests** добавьте ссылку на проект **Банк**.

В **обозревателе решений** щелкните **Зависимости** в проекте **BankTests**, а затем выберите в контекстном меню **Добавить ссылку**.

1. В диалоговом окне **Диспетчер ссылок** разверните **Проекты**, выберите **Решение** и выберите элемент **Банк**.
2. Нажмите кнопку **ОК**.

**Создание тестового класса**

Создание тестового класса, чтобы проверить класс BankAccount. Можно использовать *UnitTest1.cs*, созданный в шаблоне проекта, но лучше дать файлу и классу более описательные имена.

**Переименуйте файл и класс**

1. Чтобы переименовать файл, в **обозревателе решений** выберите файл *UnitTest1.cs* в проекте BankTests. В контекстном меню выберите команду **Переименовать**, а затем переименуйте файл в *BankAccountTests.cs*.
2. Чтобы переименовать класс, поместите курсор в UnitTest1 в редакторе кода, щелкните правой кнопкой мыши и выберите команду **Переименовать**. Введите название **BankAccountTests** и нажмите клавишу **ВВОД**.

Файл *BankAccountTests.cs* теперь содержит следующий код:

**Листинг 2**

using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace BankTests

{

[TestClass]

public class BankAccountTests

{

[TestMethod]

public void TestMethod1()

{

}

}

}

**Добавьте оператор using**

Можно также добавить [оператор using](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/using-statement) в класс, чтобы тестируемый проект можно было вызывать без использования полных имен. Вверху файла класса добавьте:

**Листинг 3**

using BankAccountNS;

**Требования к тестовому классу**

Минимальные требования к тестовому классу следующие:

* Атрибут [TestClass] является обязательным в любом классе, содержащем методы модульных тестов, которые необходимо выполнить в обозревателе тестов.
* Каждый метод теста, предназначенный для запуска в обозревателе тестов, должен иметь атрибут [TestMethod].

Можно иметь другие классы в проекте модульного теста, которые не содержат атрибута [TestClass] , а также иметь другие методы в тестовых классах, у которых атрибут — [TestMethod] . Можно вызывать эти другие классы и методы в методах теста.

**Создание первого тестового метода**

В этой процедуре мы напишем методы модульного теста для проверки поведения метода Debit класса BankAccount.

Существует по крайней мере три поведения, которые требуется проверить:

* Метод создает исключение [ArgumentOutOfRangeException](https://docs.microsoft.com/ru-ru/dotnet/api/system.argumentoutofrangeexception) , если сумма по дебету превышает баланс.
* Метод создает исключение [ArgumentOutOfRangeException](https://docs.microsoft.com/ru-ru/dotnet/api/system.argumentoutofrangeexception), если сумма по дебету меньше нуля.
* Если значение дебета допустимо, то метод вычитает сумму дебета из баланса счета.

**Совет**

Метод по умолчанию TestMethod1 можно удалять, так как он не используется в этом руководстве.

**Создание метода теста**

Первый тест проверяет, снимается ли со счета нужная сумма при допустимом размере кредита (со значением меньшим, чем баланс счета, и большим, чем ноль). Добавьте следующий метод в этот класс BankAccountTests :

**Листинг 4**

[TestMethod]

public void Debit\_WithValidAmount\_UpdatesBalance()

{

// Arrange

double beginningBalance = 11.99;

double debitAmount = 4.55;

double expected = 7.44;

BankAccount account = new BankAccount("Mr. Bryan Walton", beginningBalance);

// Act

account.Debit(debitAmount);

// Assert

double actual = account.Balance;

Assert.AreEqual(expected, actual, 0.001, "Account not debited correctly");

}

Метод очень прост: он создает новый объект BankAccount с начальным балансом, а затем снимает допустимое значение. Он использует метод [Assert.AreEqual](https://docs.microsoft.com/ru-ru/dotnet/api/microsoft.visualstudio.testtools.unittesting.assert.areequal), чтобы проверить, что конечный баланс соответствует ожидаемому.

**Требования к методу теста**

Метод теста должен удовлетворять следующим требованиям:

* Он декорируется атрибутом [TestMethod].
* Он возвращает void.
* Он не должен иметь параметров.

**Сборка и запуск теста**

1. В меню **Построение** выберите **Построить решение**.
2. Откройте **Обозреватель тестов**, выбрав **Тест** > **Windows** > **Обозреватель тестов** в верхней строке меню.
3. Выберите **Запустить все** , чтобы выполнить тест.

Во время выполнения теста в верхней части окна **Обозреватель тестов** отображается анимированная строка состояния. По завершении тестового запуска строка состояния становится зеленой, если все методы теста успешно пройдены, или красной, если какие-либо из тестов не пройдены.

В данном случае тест пройден не будет.

1. Выберите этот метод в **обозревателе тестов** для просмотра сведений в нижней части окна.

**Исправление кода и повторный запуск тестов**

Результат теста содержит сообщение, описывающее возникшую ошибку. Для метода AreEqual выводится сообщение о том, что ожидалось и что было фактически получено. Ожидалось, что баланс уменьшится, а вместо этого он увеличился на сумму списания.

Модульный тест обнаружил ошибку: сумма списания *добавляется* на баланс счета, вместо того чтобы *вычитаться*.

**Исправление ошибки**

Чтобы исправить эту ошибку, в файле *BankAccount.cs* замените строку:

**Листинг 5**

m\_balance += amount;

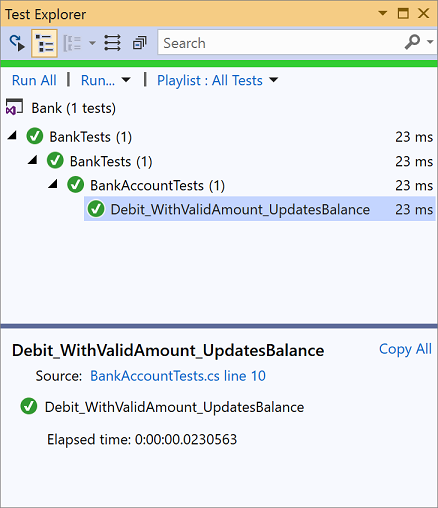
на:

**Листинг 6**

m\_balance -= amount;

**Повторный запуск теста**

В **обозревателе тестов** выберите **Запустить все**, чтобы запустить тест повторно. Красно-зеленая строка становится зеленой, чтобы указать, что тест был пройден.



**Использование модульных тестов для улучшения кода**

В этом разделе рассматривается, как последовательный процесс анализа, разработки модульных тестов и рефакторинга может помочь сделать рабочий код более надежным и эффективным.

**Анализ проблем**

Мы создали тестовый метод для подтверждения того, что допустимая сумма правильно вычитается в методе Debit. Теперь проверим, что метод создает исключение [ArgumentOutOfRangeException](https://docs.microsoft.com/ru-ru/dotnet/api/system.argumentoutofrangeexception), если сумма по дебету:

* больше баланса или
* меньше нуля.

**Создание и запуск новых методов теста**

Создадим метод теста для проверки правильного поведения в случае, когда сумма по дебету меньше нуля:

**Листинг 7**

[TestMethod]

public void Debit\_WhenAmountIsLessThanZero\_ShouldThrowArgumentOutOfRange()

{

// Arrange

double beginningBalance = 11.99;

double debitAmount = -100.00;

BankAccount account = new BankAccount("Mr. Bryan Walton", beginningBalance);

// Act and assert

Assert.ThrowsException<System.ArgumentOutOfRangeException>(() => account.Debit(debitAmount));

}

Мы используем метод [ThrowsException](https://docs.microsoft.com/ru-ru/dotnet/api/microsoft.visualstudio.testtools.unittesting.assert.throwsexception) для подтверждения правильности созданного исключения. Этот метод приводит к тому, что тест не будет пройден, если не возникнет исключения [ArgumentOutOfRangeException](https://docs.microsoft.com/ru-ru/dotnet/api/system.argumentoutofrangeexception). Если временно изменить тестируемый метод для вызова более общего исключения [ApplicationException](https://docs.microsoft.com/ru-ru/dotnet/api/system.applicationexception) при значении суммы по дебету меньше нуля, то тест работает правильно — то есть завершается неудачно.

Чтобы проверить случай, когда размер списания превышает баланс, выполните следующие действия:

1. Создать новый метод теста с именем Debit\_WhenAmountIsMoreThanBalance\_ShouldThrowArgumentOutOfRange.
2. Скопировать тело метода из Debit\_WhenAmountIsLessThanZero\_ShouldThrowArgumentOutOfRange в новый метод.
3. Присвоить debitAmount значение, превышающее баланс.

Выполните два теста и убедитесь, что они пройдены.

**Продолжение анализа**

Тестируемый метод можно дополнительно улучшить. При такой реализации мы не можем знать, какое условие (amount > m\_balance или amount < 0) приводят к исключению, возвращаемому в ходе теста. Нам просто известно, что ArgumentOutOfRangeException где-то возникает в методе. Было бы лучше знать, какое условие в BankAccount.Debit вызвало исключение (amount > m\_balance или amount < 0), чтобы быть уверенными в том, что наш метод правильно проверяет свои аргументы.

Еще раз проанализировав тестируемый метод BankAccount.Debit, можно заметить, что оба условных оператора используют конструктор ArgumentOutOfRangeException, который просто получает имя аргумента в качестве параметра:

**Листинг 8**

throw new ArgumentOutOfRangeException("amount");

Так выглядит конструктор, который можно использовать для сообщения более детальной информации: [ArgumentOutOfRangeException(String, Object, String)](https://docs.microsoft.com/ru-ru/dotnet/api/system.argumentoutofrangeexception.-ctor" \l "System_ArgumentOutOfRangeException__ctor_System_String_System_Object_System_String_) включает имя аргумента, значения аргумента и определяемое пользователем сообщение. Мы можем выполнить рефакторинг тестируемого метода для использования данного конструктора. Более того, можно использовать открытые для общего доступа члены типа для указания ошибок.

**Рефакторинг тестируемого кода**

Сначала определим две константы для сообщений об ошибках в области видимости класса. Добавьте это в тестируемый класс (BankAccount):

**Листинг 9**

public const string DebitAmountExceedsBalanceMessage = "Debit amount exceeds balance";

public const string DebitAmountLessThanZeroMessage = "Debit amount is less than zero";

Затем изменим два условных оператора в методе Debit:

**Листинг 10**

if (amount > m\_balance)

{

throw new System.ArgumentOutOfRangeException("amount", amount, DebitAmountExceedsBalanceMessage);

}

if (amount < 0)

{

throw new System.ArgumentOutOfRangeException("amount", amount, DebitAmountLessThanZeroMessage);

}

**Рефакторинг тестовых методов**

Выполните рефакторинг методов теста, удалив вызов [Assert.ThrowsException](https://docs.microsoft.com/ru-ru/dotnet/api/microsoft.visualstudio.testtools.unittesting.assert.throwsexception). Заключите вызов Debit() в блок try/catch, перехватите конкретное ожидаемое исключение и проверьте соответствующее ему сообщение. Метод [Microsoft.VisualStudio.TestTools.UnitTesting.StringAssert.Contains](https://docs.microsoft.com/ru-ru/dotnet/api/microsoft.visualstudio.testtools.unittesting.stringassert.contains) обеспечивает возможность сравнения двух строк.

В этом случае метод Debit\_WhenAmountIsMoreThanBalance\_ShouldThrowArgumentOutOfRange может выглядеть следующим образом:

**Листинг 11**

[TestMethod]

public void Debit\_WhenAmountIsMoreThanBalance\_ShouldThrowArgumentOutOfRange()

{

// Arrange

double beginningBalance = 11.99;

double debitAmount = 20.0;

BankAccount account = new BankAccount("Mr. Bryan Walton", beginningBalance);

// Act

try

{

account.Debit(debitAmount);

}

catch (System.ArgumentOutOfRangeException e)

{

// Assert

StringAssert.Contains(e.Message, BankAccount.DebitAmountExceedsBalanceMessage);

}

}

**Повторное тестирование, переписывание и анализ**

Метод теста сейчас обрабатывает не все требуемые случаи. Если тестируемый метод Debit не смог выдать исключение [ArgumentOutOfRangeException](https://docs.microsoft.com/ru-ru/dotnet/api/system.argumentoutofrangeexception), когда значение debitAmount было больше остатка (или меньше нуля), метод теста выдает успешное прохождение. Это нехорошо, поскольку метод теста должен был завершиться с ошибкой в том случае, если исключение не создается.

Это является ошибкой в методе теста. Для решения этой проблемы добавим утверждение [Fail](https://docs.microsoft.com/ru-ru/dotnet/api/microsoft.visualstudio.testtools.unittesting.assert.fail) в конце тестового метода для обработки случая, когда исключение не создается.

Однако повторный запуск теста показывает, что тест теперь оказывается *непройденным* при перехватывании верного исключения. Блок catch перехватывает исключение, но метод продолжает выполняться, и в нем происходит сбой на новом утверждении [Fail](https://docs.microsoft.com/ru-ru/dotnet/api/microsoft.visualstudio.testtools.unittesting.assert.fail). Чтобы разрешить эту проблему, добавим оператор return после StringAssert в блоке catch. Повторный запуск теста подтверждает, что проблема устранена. Окончательная версия метода Debit\_WhenAmountIsMoreThanBalance\_ShouldThrowArgumentOutOfRange выглядит следующим образом:

**Листинг 12**

[TestMethod]

public void Debit\_WhenAmountIsMoreThanBalance\_ShouldThrowArgumentOutOfRange()

{

// Arrange

double beginningBalance = 11.99;

double debitAmount = 20.0;

BankAccount account = new BankAccount("Mr. Bryan Walton", beginningBalance);

// Act

try

{

account.Debit(debitAmount);

}

catch (System.ArgumentOutOfRangeException e)

{

// Assert

StringAssert.Contains(e.Message, BankAccount.DebitAmountExceedsBalanceMessage);

return;

}

Assert.Fail("The expected exception was not thrown.");

}

**Заключение**

Усовершенствования тестового кода привели к созданию более надежных и информативных методов теста. Но что более важно, в результате был также улучшен тестируемый код.

# Контрольная работа №1

Цель: Формирование модели АИС (TO BE): научиться моделировать, анализировать и совершенствовать бизнес-процессы, разрабатывать и анализировать архитектуру предприятия

Задание: Для выбранного предприятия, связанного с темой НИР, на основе имеющихся данных сформировать TO BE:

* бизнес-модель в рамках структурного подхода в виде декомпозиции IDEF0 диаграмм.
* Описание потоков данных DFD.

В случае выбора объектно-ориентированного подхода формируется полный комплект диаграмм UML

# Контрольная работа №2

Цель: Описание списка документов АИС и Форм этих документов.

Задание: Для выбранного предприятия, связанного с темой НИР, на основе имеющихся данных уточнить описанные на 1 этапе ЖЦ:

1. Список документов, сгруппировав по видам:
   1. Входные документы
      1. Нормативно-справочные документы (НСД)
      2. Оперативного учета
   2. Выходные документы
2. Описание потоков данных в виде отдельного дерева декомпозиции DFD диаграмм (всего 2 уровня).
3. Формы (бланки, шаблоны) всех документов из списка

# Контрольная работа №3

Цель: сформировать проект БД и провести **автоматизированный** переход к 3 этапу ЖЦ – Реализации (в плане создания БД в выбранной СУБД)

Задание: Для выбранного предприятия, связанного с темой НИР, на основе имеющихся данных сформировать проект БД и провести автоматизированный переход к 3 этапу ЖЦ – Реализации (в плане создания БД в выбранной СУБД) по аналогии с:

**Лабораторной №1.**

Создается логически-физическая модель. Для выбранного предприятия, связанного с темой НИР, и в соответствии с Контрольной работой №2 создается описание логической модели БД.

**Лабораторной №2.**

Создается описание физической модели БД из Лабораторной №1 в соответствии с выбранной СУБД

**Лабораторной №3.**

провести **автоматизированный** переход к 3 этапу ЖЦ – Реализации, т.е. создать БД в выбранной СУБД либо методом прямого проектирования через connect с ней, либо через SQLскрипт.

# Контрольная работа №4

Цель: Подготовить тест кейсы для АИС.

Задание: Для выбранного предприятия, связанного с темой НИР, на основе имеющихся данных сформировать тест кейсы по аналогии с:

**Лабораторной №1.**

Для тестирования белого ящика

**Лабораторной №2.**

Для тестирования черного ящика

**Лабораторной №3.**

Для модульного тестирования

# Список литературы

1. Косяков Александр, Уильям Н. Свит, Сэмюэль Дж. Сеймур, Стивен М. Бимер., «Системная инженерия. Принципы и практика» – М.: ДМК Пресс, 2014.
2. Мышенков К.С., Беляшов А.Н., «Классификация методов анализа и проектирования систем управления» // Математическое моделирование и информатика: Труды XV науч. конф. / Под ред. Д.Ю. Рязанова. – М.: ИЦ ФГБОУ ВПО МГТУ «СТАНКИН», 2013. – С. 45-47.
3. [Анализ требований к автоматизированным информационным системам](https://intuit.ru/studies/courses/2188/174/info): <https://intuit.ru/studies/courses/2188/174/lecture/4714?page=1>
4. ISO/IEC 12207:2008 Systems and Software Development - Software Life Cycle Processes. [Electronic resource]. URL: http://www.iso.org/iso/catalogue\_detail?csnumber=43447
5. ГОСТ Р ИСО/МЭК 12207-2010. Информационная технология. Системная и программная инженерия. Процессы жизненного цикла программных средств URL: http://docs.cntd.ru/document/gost-r-iso-mek-12207-2010
6. R 50.1.028-2001 methodology of functional modeling [Electronic resource]. URL: https://docinfo.ru/r/r-50-1-028-2001/.
7. D.Yu. Kiselev, Yu.V. Kiselev, V.D. Makariev. Structural analysis of data flows (Data flow diagrams - DFD) Samara: SGAU Publishing House, 2014. 12 с. URL: http://repo.ssau.ru/bitstream/Metodicheskie-ukazaniya/Strukturnyi-analiz-potokov-dannyh-Data-Flow-DiagramsDFD-Elektronnyi-resurs-metod-ukazaniya-53592/1/Киселев%20Д.Ю.%20Структурный%20анализ.pdf
8. ГОСТ 34.602-89. Информационная технология. Комплекс стандартов на автоматизированные системы. Техническое задание на создание автоматизированной системы. URL:http://docs.cntd.ru/document/gost-34-602-89
9. Лоусон Г., Путешествие по системному ландшафту. — ДМК-Пресс, 2013.
10. Мельников В.П., Информационное обеспечение систем управления – М.: ACADEMIA, 2010.
11. <https://de.donstu.ru/elect/divisions/679/courses>