*МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ*

федеральное государственное бюджетное образовательное учреждение
высшего образования
«Курганский государственный университет»

Кафедра программного обеспечения автоматизированных систем

УПРАВЛЕНИЕ КАЧЕСТВОМ И ТЕСТИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Методические указания
к выполнению контрольных работ
для студентов направлений 09.03.03, 09.03.04

Курган 2026

Кафедра: «Программное обеспечение автоматизированных систем»

Дисциплина: «Управление качеством и тестирование ПО»

Направление подготовки: 09.03.03 «Прикладная информатика»,
09.03.04 «Программная инженерия»

Составил: канд. техн. наук, доцент А. М. Семахин.

Печатается в соответствии с планом издания, утверждённым методическим советом университета «17» декабря 2025 г.

Утверждены на заседании кафедры « 29 » декабря 2025 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ 4
1 ТЕСТИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ 4
1.1 Основные понятия и определения 4
2 СТРУКТУРНОЕ ТЕСТИРОВАНИЕ 5
2.1 Критерии структурного тестирования 6
2.2 Методы структурного тестирования 6
2.3 Достоинства и недостатки тестирования «белого ящика» 10
3 ФУНКЦИОНАЛЬНОЕ ТЕСТИРОВАНИЕ 10
3.1 Функциональные критерии 12
3.2 Методы функционального тестирования 12
4 ИНТЕГРАЦИОННОЕ ТЕСТИРОВАНИЕ 20
4.1 Методы интеграционного тестирования 21
4.1.1 Нисходящее тестирование 21
4.1.2 Восходящее тестирование 23
4.1.3 Сравнение нисходящего и восходящего тестирования 23
5 КОНТРОЛЬНАЯ РАБОТА 25
5.1 Назначение, цели и задачи контрольной работы 25
5.2 Требования к контрольной работе 25
5.2.1 Требования к функциональным характеристикам 25
5.2.2 Требования к эксплуатационным характеристикам 26
5.2.3 Требования к программному обеспечению 26
5.2.4 Варианты заданий контрольной работы 26
ЗАКЛЮЧЕНИЕ 31
БИБЛИОГРАФИЧЕСКИЙ СПИСОК 32

ВВЕДЕНИЕ

Дисциплина «Управление качеством и тестирование программного обеспечения» имеет цель дать студентам теоретические знания и практические навыки в выявлении ошибок в программных продуктах.

Предмет дисциплины – технология тестирования программных приложений.

Задачи дисциплины – дать представление о качестве и надёжности программного обеспечения, изучить методы тестирования программных продуктов, сформировать навыки тестирования программных приложений.

Методические указания содержат теоретическое обоснование и варианты заданий для выполнения контрольных работ по дисциплине «Управление качеством и тестирование программного обеспечения».

Методические указания разработаны в соответствии с требованиями государственного образовательного стандарта по подготовке бакалавров по направлениям 09.03.03 «Прикладная информатика» и 09.03.04 «Программная инженерия».

1 ТЕСТИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Динамическим методом верификации программного обеспечения является тестирование программы.

1.1 Основные понятия и определения тестирования

Тестирование (testing) – процесс обнаружения ошибок в программе.

Отладка (debugging) – процесс обнаружения и исправления ошибок в программе.

Контроль (verification) – процесс обнаружения ошибки при выполнении программы в тестовой или моделируемой среде.

Испытание (validation) – процесс обнаружения ошибки при выполнении программы в заданной реальной среде.

Аттестация (certification) – авторитетное подтверждение правильности программы.

Тестирование модуля (module testing) – контроль отдельного программного модуля.

Тестирование сопряжений (integration testing) – контроль сопряжений между частями системы (модулями, компонентами, подсистемами).

Тестирование внешних функций (external function testing) – контроль внешнего поведения системы, определённого внешними спецификациями.

Комплексное тестирование (system testing) – контроль и/или испытание системы по отношению к исходным целям.

Тестовый случай (test case) – набор входных данных, на которых программа исполняется в процессе тестирования.

Тестирование приемлемости (acceptance testing) – проверка соответствия программы требованиям пользователя.

Тестирование настройки (installation testing) – процесс выявления ошибок, возникших в процессе настройки системы [1].

2 СТРУКТУРНОЕ ТЕСТИРОВАНИЕ

Структурное тестирование – процесс обнаружения ошибок в логике программы.

Покрытие кода тестами – мера, показывающая, на сколько процентов исходный код программы был протестирован.

Стратегия «белого ящика», или стратегия тестирования, управляемого логикой (текста) программы, позволяет исследовать внутреннюю структуру программы. Тестирование заключается в проверке каждого пути, каждой ветви алгоритма. Внешняя спецификация во внимание не принимается. Тестирующий получает тестовые данные посредством анализа программы [1].

Известны: внутренняя структура программы.

Исследуются: внутренние элементы программы и связи между ними (рисунок 2.1).

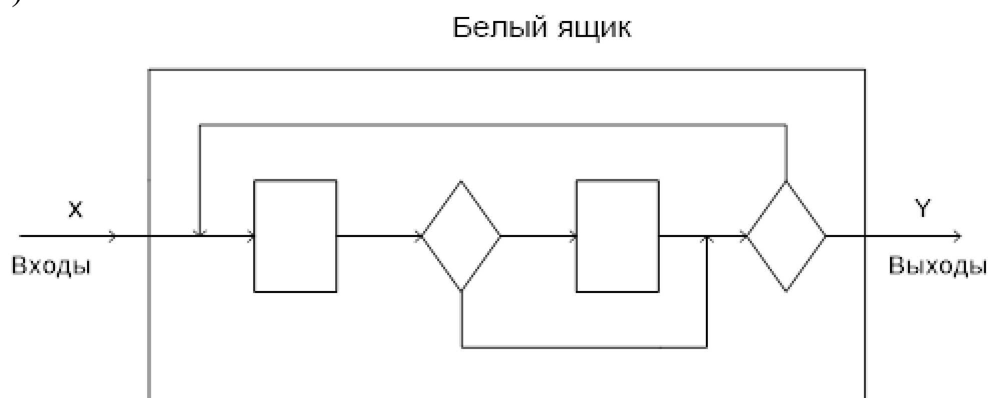

Рисунок 2.1 – Внутренняя структура программы

Особенности тестирования «белого ящика».

Обычно тестирование «белого ящика» основано на анализе управляющей структуры программы. Программа считается полностью проверенной, если проведено исчерпывающее тестирование маршрутов (путей) ее графа управления.

Для этого формируются тестовые варианты, в которых:

- гарантируется проверка всех независимых маршрутов программы;
- проходятся ветви *True False* для всех логических решений;
- выполняются все циклы (в пределах границ и диапазонов);
- анализируется правильность внутренних структур данных.

2.1 Критерии структурного тестирования

Структурные критерии используют модель программы в виде «белого ящика». Структурные критерии базируются на основных элементах управляющего потокового графа, операторах, ветвях и путях:

- критерий тестирования команд (критерий *C0)* – набор тестов, обеспечивающий прохождение команды не менее одного раза;
- критерий тестирования ветвей (критерий *C1*) – набор тестов, обеспечивающий прохождение каждой ветви не менее одного раза.
- критерий тестирования путей (критерий *C2*) – набор тестов, обеспечивающий прохождение каждого пути не менее одного раза [2-11].

2.2 Методы структурного тестирования

Методы структурного тестирования:

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

Метод покрытия операторов выполняет каждый оператор хотя бы один раз. Покрытие операторов – слабый критерий, так как выполнение оператора хотя бы один раз есть необходимое, но недостаточное условие для тестирования по стратегии «белого ящика».

Метод покрытия решений выполняет каждое направление перехода хотя бы один раз. Покрытие решений (покрытие переходов) – сильный критерий покрытия логики программы. Покрытие решений удовлетворяет критерию покрытия операторов. При выполнении каждого направления перехода оператор должен быть выполнен.

Исключения составляют:

- программы, не имеющие решений;
- программы с несколькими точками входа;
- операторы переключатели *switch*.

Покрытие решений требует, чтобы каждое решение имело результатом значение истина или ложь и каждый оператор выполнялся, по крайней мере, один раз.

Метод покрытия условий выполняет каждое условие хотя бы один раз. Покрытие условий – сильный критерий покрытия логики программы. Покрытие условий удовлетворяет критерию покрытия решений. Покрытие условий предусматривает число тестов, достаточное для того, чтобы все возможные результаты каждого условия в решении выполнялись, по крайней мере, один раз. Каждой точке входа в программу, а также *switch-операторам* должно быть передано управление при вызове, по крайней мере, один раз.

Метод покрытия решений/условий выполняет условия и результаты решения, по крайней мере, один раз и каждой точке входа передаётся управление, по крайней мере, один раз. Покрытие решений/условий требует набора тестов, чтобы все возможные результаты каждого условия в решении, все результаты каждого решения выполнялись, по крайней мере, один раз и каждой точке входа передавалось управление, по крайней мере, один раз. Покрытие решений/условий невозможно применить для выполнения всех результатов всех условий вследствие того, что определённые условия скрыты другими условиями.

Метод комбинаторного покрытия условий выполняет покрытия решений, покрытия условий и покрытия решений/условий. Комбинаторное покрытие условий требует создания набора тестов, чтобы все возможные комбинации результатов условия в каждом решении и все точки входа выполнялись, по крайней мере, один раз [12-14].

Метод базового пути позволяет:

- получить оценку комплексной сложности программы;
- использовать эту оценку для определения необходимого количества тестовых вариантов.

Тестовые варианты гарантируют однократное выполнение каждого оператора программы при тестировании.

Для представления программы используется потоковый граф. Особенности потокового графа:

1 Граф отображает управляющую структуру программы. Закрывающие скобки условных операторов и циклов *(end if, end loop)* рассматриваются как отдельные (фиктивные) операторы.

2 Узел (вершина) потокового графа соответствуют линейным участкам программы и включают один или несколько операторов.

3 Дуги (ориентированные ребра) потокового графа отображают поток управления в программе (передача управления между операторами).

4 Различают операторные и предикатные узлы. Из операторного узла выходит одна дуга, а из предикатного – две дуги.

5 Предикатные узлы соответствуют простым условиям в программе. Составным называется условие, в котором используются булевы операции *(OR, AND)*. Например, фрагмент программы:

if a OR b then x else y; z;

где *x, y* и *z* – операторы присваивания

будет представлен потоковым графом (рисунок 2.2):

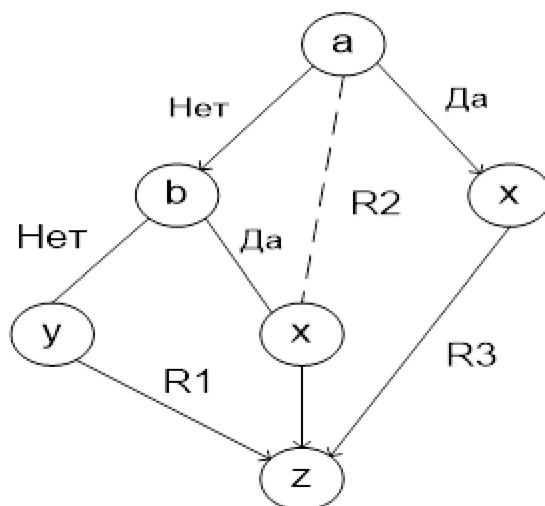

Рисунок 2.2 – Потоковый граф

1 Замкнутые области, образованные дугами и узлами, называются регионами.

2 Окружающая среда рассматривается как исполнительный регион. Рассмотренный граф имеет три региона – *R1, R2, R3*.

Цикломатическая сложность – это матрица ПО, которая обеспечивает количественную оценку логической сложности программы. В способе тестирования базового пути цикломатическая сложность определяет:

- количество независимых путей в базовом множестве программы;
- верхнюю оценку количества тестов, которая гарантирует однократное выполнение всех операторов.

Независимый путь – путь, который вводит новый оператор обработки или новое условие (содержит дугу, которая не входит в ранее определенные пути).

Все независимые пути графа образуют базовое множество. Свойства базового множества:

1 тесты, обеспечивающие его проверку, гарантируют:

- однократное выполнение каждого оператора;
- выполнение каждого условия по *true*-ветви и *false*-ветви;

2 мощность базового множества равна цикломатической сложности потокового графа.

Рассмотрим шаги способа тестирования базового пути на примере процедуры вычисления среднего значения положительных чисел массива.

```
s = 0;   kp = 0;    cin  >> n;
if (n > 0)
{for (I = 0;  I < n;  i++)
cin >> m[i];
for (I = 0;  I < n;  i++)
```

```
if (m[i] > 0) { s += m[i];
kp++; }        (11)        (12)
if (kp > 0) sr = s / kp;
                                    (13)
else sr = 0;
(14)
cout << sr;}
```

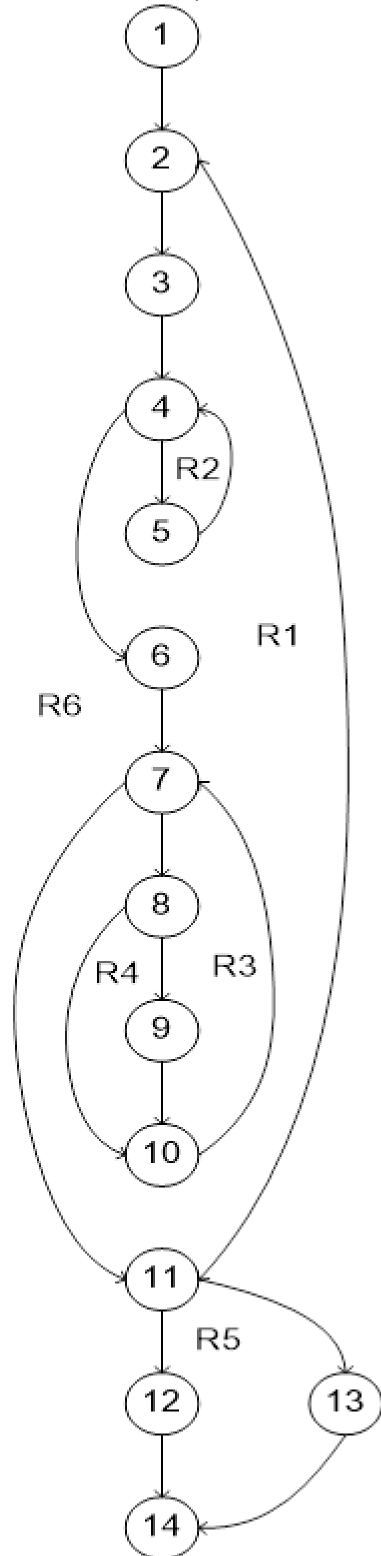

Рисунок 2.3 – Управляющий потоковый граф

Шаг 1 На основе программы формируется потоковый граф.

Шаг 2 Определение цикломатической сложности потокового графа:

1) V(G) = 6 регионов;

2) V(G) = 18 дуг – 14 уз + 2 = 6;

3) V(G) = 5 пред. уз. + 1 = 6.

Шаг 3 Определяется базовое множество независимых линейных путей.

Путь 1: 1 – 2 – 11 – 13 – 14

Путь 2: 1 – 2 – 11 – 12 – 14

Путь 3: 1 – 2 – 3 – 4 – 6 – 7 -11 – 12 – 14

Путь 4: 1 – 2 – 3 – 4 – 6 – 7 – 11 – 13 – 14

Путь 5: 1 – 2 – 3 – 4 – 5 – 4 – 6 – 7 – 11 – 12 – 14

Шаг 4 Подготавливаются тестовые варианты, инициирующие выполнение каждого пути. Каждый тестовый вариант формируется в следующем виде:

Исходные данные (ИД);

Ожидаемые результаты (Ож. РЕЗ).

Тестовый вариант для пути 1 ТВ1:

ИД n = 0 или n < 0

Ож. РЕЗ Sr = 0

Тестовый вариант для пути 2 ТВ2:

Невозможен. Проверка узла 12 будет выполнена в пути 3.

Управляющий потоковый граф представлен на рисунке 2.3.

Цикломатическая сложность вычисляется одним из трех способов:

1 равна количеству регионов потокового графа;

2 определяется по формуле $V(G) = E - N + 2$, где E – количество дуг, N – количество узлов потокового графа;

3 определяется по формуле $V(G) = p + 1$, где p – количество предикатных узлов в потоковом графе.

2.3 Достоинства и недостатки тестирования «белого ящика»

Достоинства тестирования «белого ящика» связаны с тем, что позволяют учесть особенности программных ошибок:

1 Количество ошибок минимально в «центре» и максимально на «периферии» программы.

2 Предварительные предположения о вероятности потока управления или данных в программе часто бывают некорректны. В результате типовым может стать маршрут, модель вычисления по которому проработана слабо.

3 При записи алгоритма программного обеспечения в виде текста на языке программирования возможно внесение типовых ошибок трансляции (синтаксических и семантических).

4 Некоторые результаты в программе зависят не от исходных, а от внутренних состояний программы.

Тесты «черного ящика» не реагируют на ошибки таких типов.

Недостатки тестирования «белого ящика»:

1 количество независимых маршрутов может быть очень велико. Например, если цикл выполняется k раз, а внутри цикла имеется n ветвлений, то количество маршрутов вычисляется по формуле:

$$m = \sum_{i=1}^{k} n^i,$$

Если $n = 5$ и $k = 20$, то $m = 10^{14}$.

2 исчерпывающее тестирование маршрутов не гарантирует соответствия программы исходным требованиям к ней;

3 в программе могут быть пропущены некоторые маршруты;

4 невозможно обнаружить ошибки, появление которых зависит от обрабатываемых данных *(if abs(a-b) < ep)*.

3 ФУНКЦИОНАЛЬНОЕ ТЕСТИРОВАНИЕ

Функциональное тестирование – процесс обнаружения ошибок в функциях программы на всей области определения.

Тесты проектируются на основе внешних спецификаций программ и модулей, или спецификаций сопряжения модуля с другими модулями, программа рассматривается как «чёрный ящик». Тест проверяет соответствие программы внешним спецификациям. Содержание программного модуля не имеет значения.

Известны: функции программы.

Исследуется: работа каждой функции на всей области определения (рисунок 3.1).

«черный ящик»

x → F(x) → y

Входы Выходы

Рисунок 3.1 –Тестирование «чёрного ящика»

Тесты демонстрируют:
- как выполняются функции программы;
- как принимаются исходные данные;
- как вырабатываются исходные данные;
- как вырабатываются результаты;
- как сохраняется целостность внешней информации.

Тестирование «черного ящика» (функциональное тестирование) позволяет получить комбинацию входных данных, обеспечивающих полную проверку всех функциональных требований к программе. При таком подходе желательно иметь:

- набор, образуемый такими входными данными, которые приводят к аномалиям поведения программы (назовем его *IT*);

- набор, образуемый такими выходными данными, которые демонстрируют дефекты программы (назовем его *OT*) (рисунок 3.2).

Входные данные IT

Программное изделие

Выходные данные OT

Рисунок 3.2 – Наборы данных

Принцип «черного ящика» не альтернативен принципу «белого ящика», поскольку он обнаруживает другой план ошибок.

Тестирование «черного ящика» обеспечивает поиск следующих категорий ошибок:

1) некорректных или отсутствующих функций;
2) ошибок интерфейса;
3) ошибок во внешних структурах данных или доступе к внешней базе данных;
4) ошибок характеристик (необходимость памяти и т. п.);
5) ошибок инициализации и завершения.

Эти категории ошибок способами «белого ящика» не выявляются.

В отличие от тестирования «белого ящика» тестирование «черного ящика» производится на поздних стадиях тестирования.

Техника «черного ящика» ориентирована на решение следующих задач:

- сокращение необходимого количества тестовых вариантов (из-за проверки не статических, а динамических аспектов системы);
- выявление классов ошибок, а не отдельных ошибок.

Исчерпывающее тестирование, как правило, невозможно, тестирование не реагирует на многие особенности программных ошибок.

3.1 Функциональные критерии

Функциональный критерий обеспечивает контроль степени выполнения требований заказчика в программном продукте. Функциональные критерии подразделяются на виды:

- тестирование проектов спецификации – набор тестов, обеспечивающий проверку каждого тестируемого пункта не менее одного раза;
- тестирование классов входных данных – набор тестов, обеспечивающий проверку представителя каждого класса входных данных не менее одного раза;
- тестирование правил – набор тестов, обеспечивающий проверку каждого правила, если входные и выходные значения описываются набором правил грамматики;
- тестирование классов выходных данных – набор тестов, обеспечивающий проверку представителя каждого выходного класса, при условии, что выходные результаты заранее расклассифицированы, отдельные классы результатов учитывают ограничения на ресурсы или на время *(time out)*;
- тестирование функций – набор тестов, обеспечивающий проверку каждого действия, реализуемого тестируемым модулем не менее одного раза;
- комбинированные критерии для программ и спецификаций – набор тестов, обеспечивающий проверку всех комбинаций непротиворечивых условий программ и спецификаций не менее одного раза [7, 12-14].

3.2 Методы функционального тестирования

Методы функционального тестирования:

- эквивалентное разбиение (классы эквивалентности);

- анализ граничных значений;
- метод функциональных диаграмм;
- предположение об ошибке.

Эквивалентное разбиение – метод функционального тестирования программы, основанный на разбиении области определения исходных данных функции на классы эквивалентности.

Класс эквивалентности – набор данных с общими свойствами.

Положения метода эквивалентного разбиения классов:

- исходные данные разбиваются на конечное число классов эквивалентности. В одном классе эквивалентности содержатся такие тесты, что если один тест из класса эквивалентности обнаруживает некоторую ошибку, то и любой другой тест из этого класса эквивалентности должен обнаруживать эту же ошибку;
- каждый тест должен включать максимальное количество классов эквивалентности, чтобы минимизировать общее число тестов.

Разработка тестов методом эквивалентного разбиения классов производится двумя этапами:

- выделение классов эквивалентности;
- построение теста.

Классы эквивалентности выделяются посредством выбора каждого входного условия, которые выбираются из технического задания или спецификации и разбиваются на две и более группы (таблица 3.1).

Таблица 3.1 – Классы эквивалентности

Входные условия	Правильные классы эквивалентности	Неправильные классы эквивалентности

Определение классов эквивалентности осуществляется эвристическим способом.

Правила определения классов эквивалентности:

- если входное условие задаёт диапазон значений, то определяют один правильный класс эквивалентности и два неправильных;
- если входное условие задаёт конкретное значение, то определяют один правильный и два неправильных класса эквивалентности;
- если входное условие задаёт множество значений, то определяют один правильный и один неправильный классы эквивалентности;
- если входное условие задаёт булево значение, то определяют один правильный и один неправильный классы эквивалентности.

После определения классов эквивалентности разрабатываются тестовые варианты. Тестовый вариант выбирается таким образом, чтобы проверить наибольшее количество свойств классов эквивалентности.

Определение тестовых вариантов:

- Каждому классу эквивалентности присваивается уникальный номер.
- Если еще остались не включенные в тесты правильные классы, то пишутся тесты, которые покрывают максимально возможное количество классов.
- Если остались не включенные в тесты неправильные классы, то пишут тесты, которые покрывают только один класс.

Анализ граничных значений – метод функционального тестирования программного обеспечения, основанный на создании тестовых вариантов, анализирующих граничные значения. Обнаружение ошибок производится на границах области ввода данных. Анализ граничных значений дополняет метод классов эквивалентности.

Отличительные особенности анализа граничных значений от эквивалентного разбиения:

- тестовые варианты создаются для проверки рёбер классов эквивалентности;
- при создании тестовых вариантов учитывают не только условия ввода, но и области вывода.

Метод требует определённой степени творчества и специализации в рассматриваемой задаче.

Правила анализа граничных значений:

- если условие ввода задаёт диапазон значений, то тестовые варианты определяются для значений чуть левее нижней границы диапазона и значения чуть правее верхней границы диапазона;
- если условие ввода задаёт дискретное множество значений, то создаются тестовые варианты для проверки минимального и максимального значений и для значений чуть меньше минимума и чуть больше максимума;
- для условий вывода применяются правила 1 и 2. Например, если необходимо вывести таблицу значений, где количество строк и столбцов меняется, то задаётся тестовый вариант для минимального вывода и тестовый вариант для максимального вывода;
- если внутренние структуры данных программы имеют предписанные границы, то разрабатываются тестовые варианты, проверяющие структуры на границах.
- если входные или выходные данные программы являются упорядоченными множествами, то тестируют первый и последний элементы множеств.

Преимущество анализа граничных значений – обнаружение большого числа ошибок.

Недостатки анализа граничных значений:

- определение границ для задачи является трудоёмким процессом;
- отсутствие проверки комбинации входных значений.

Пример. Протестировать программу бинарного поиска. Поиск выполняется в массиве *M*, возвращается индекс *I* элемента массива, значение которого равно ключу поиска *K*.

Предусловия:

1) массив упорядочен;
2) массив имеет не менее одного элемента;
3) нижняя граница массива меньше или равна его верхней границе.

Постусловия:

1) если элемент найден, то флаг *Result = True*, *I* – номер элемента;
2) если элемент не найден, то флаг *Result = False*, *I* – не определено.

Для формирования классов эквивалентности (и их ребер) нужно произвести разбиение области исходных данных – построить дерево разбиения. Листья дерева дадут искомые классы эквивалентности.

На первом уровне анализируем выполнимость предусловий, на втором уровне – выполнимость предусловий, на третьем – специальные требования, полученные из практики разработчика (рисунок 3.3).

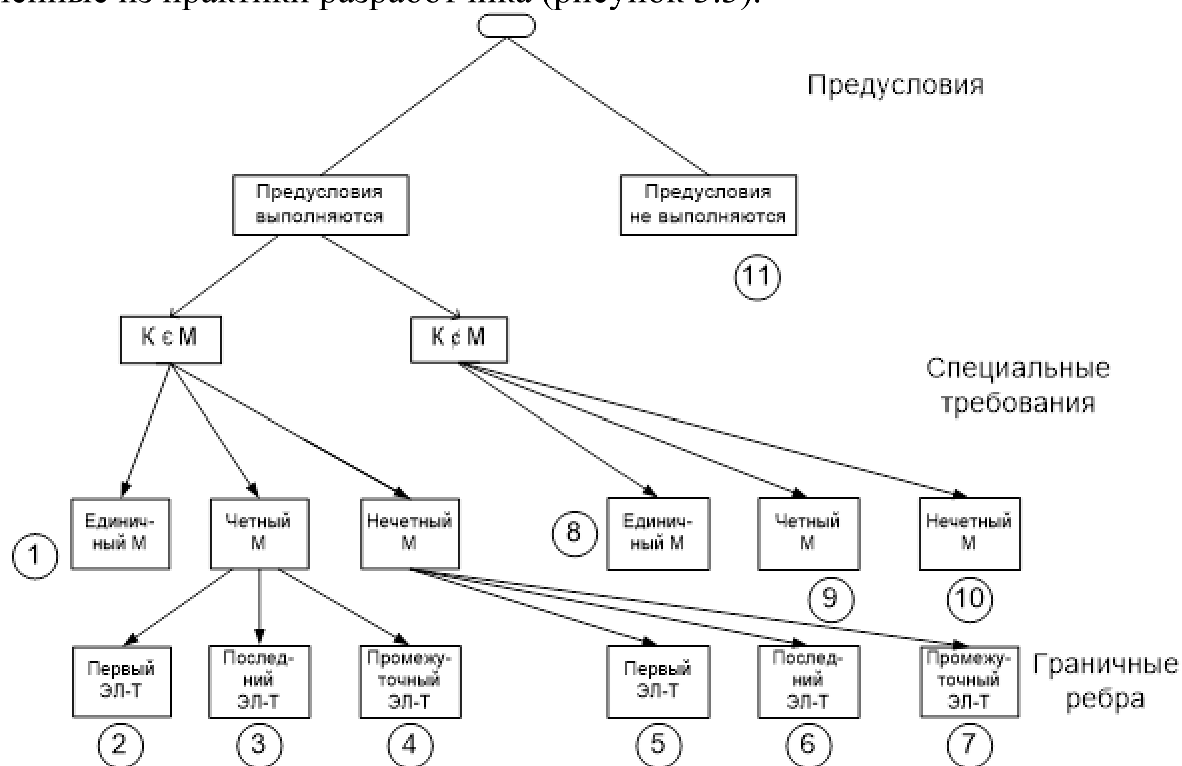

Рисунок 3.3 – Дерево разбиений области данных бинарного поиска

Дерево имеет 11 листьев. Каждый лист определяет отдельный тестовый вариант.

Примеры тестовых вариантов (ТВ).

ТВ 1:

ИД : М = 10; К = 10.

ОЖ.РЕЗ : Result = True; I = 1

ТВ 2:

ИД : М = 2 7 10 17 28 40; К = 2;

ОЖ.РЕЗ : Result = True; I = 1

ТВ 6:

ИД : М = 2 7 10 17 28 40 51; Л = 51

ОЖ.РЕЗ : Result = True; I = 7

ТВ 9:

ИД : М = 2 7 10 17 28 40;

ОЖ.РЕЗ :Result = False ; I = ?

ТВ 11:

ИД : М = 28 2 10 17 40 7; К = 2^8

ОЖ.РЕЗ : Аварийное сообщение : массив не упорядочен

Метод функциональных диаграмм – метод функционального тестирования программного обеспечения, основанный на проектировании тестовых вариантов, использующих формальную запись логических условий и соответствующих действий.

Этапы метода функциональных диаграмм:

1 для каждого модуля перечисляются причины (условия ввода, классы эквивалентности условий ввода) и следствия (действия или условия вывода). Идентификатор присваивается причине и следствию;

2 разрабатывается диаграмма причинно-следственных связей;

3 определяется таблица решений на основе диаграммы причинно-следственных связей;

4 столбцы таблицы решений преобразуются в тестовые варианты.

Таблица решений снабжается примечаниями, задающими ограничения и описывающими комбинации, которые невозможны.

Преимущество метода функциональных диаграмм – наглядность диаграммы причинно-следственных связей.

Недостаток метода функциональных диаграмм – плохое исследование граничных условий.

Причины будем обозначать символами c_i, а следствия – e_i.

Каждый узел графа может находиться в состоянии 0 или 1 (0 – состояние отсутствует, 1 – присутствует).

Функция тождество устанавливает, что если значение c_1 есть 1, то и значение e_1 есть 1, в противном случае значение e_1 есть 0 (рисунок 3.4).

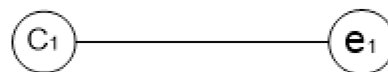

Рисунок 3.4 – Функция «Тождество»

Функция не устанавливает, что если значение c_1 есть 1, то значение e_1 есть 0; в противном случае значение e_1 есть 0 (рисунок 3.5).

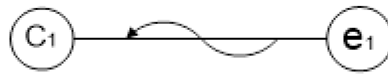

Рисунок 3.5 – Функция «Не»

Функция или устанавливает, что если c_1 или c_2 есть 1, то e_1 есть 1, в противном случае e_1 есть 0 (рисунок 3.6).

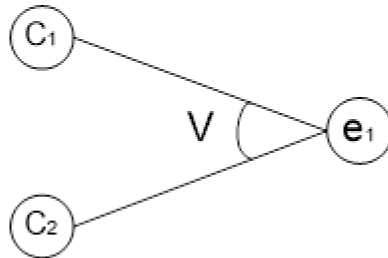

Рисунок 3.6 – Функция «ИЛИ»

Функция и устанавливает, что если c_1 и c_2 есть 1, то e_1 есть 1, в противном случае e_1 есть 0 (рисунок 3.7).

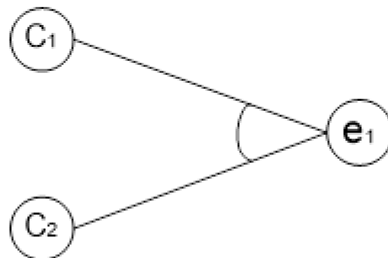

Рисунок 3.7 – Функция «И»

Ограничение E (Exclusive – исключает) устанавливает, что E должно быть истинным, если хотя бы одна из величин – *a* или *b* – принимает значение 1 (*a* и *b* не могут принимать значение 1 одновременно) (рисунок 3.8).

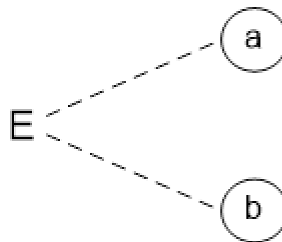

Рисунок 3.8 – Ограничение «Exclusive»

Ограничение I (Inclusive – включает) устанавливает, что по крайней мере одна из величин *a, b* или *c* всегда должна быть равной 1 (*a, b* и *c* не могут принимать значение 0 одновременно) (рисунок 3.9).

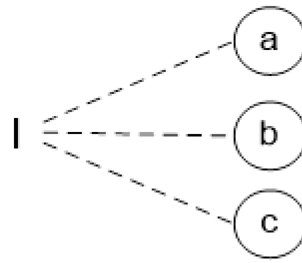

Рисунок 3.9 – Ограничение «Inclusive»

Ограничение O (Only one – одно и только одно) устанавливает, что одна и только одна из величин *a* или *b* должна быть равна 1 (рисунок 3.10):

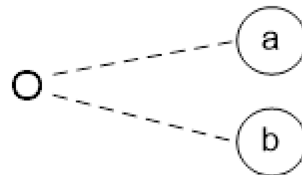

Рисунок 3.10 – Ограничение «Only one»

Ограничение R (Requires – требует) устанавливает, что если *a* принимает значение 1, то и *b* должна принимать значение 1 (нельзя, чтобы *a* было равно 1, *b* – 0) (рисунок 3.11).

Рисунок 3.11 – Ограничение «Requires»

Ограничение M (Masks – скрывает,) устанавливает, что если следствие *a* имеет значение 1, то следствие *b* должно принять значение 0 (рисунок 3.12):

Рисунок 3.12 – Ограничение «Masks»

Пример. Программа выполняет расчет оплаты за электроэнергию по среднему или переменному тарифу.

При расчете по среднему тарифу:

- при месячном потреблении энергии меньшем, чем 100 квт/ч. вычисляется фиксированная сумма;

- при потреблении энергии, большем или равном 100 квт/ч., применяется процедура А планирования расчета.

При расчете по переменному тарифу:

- при месячном потреблении энергии меньше 100 квт/ч. применяется процедура А планирования расчета;

- при потреблении энергии, большем или равном 100 квт/ч., применяется процедура В планирования расчета.

Шаг 1. Причинами являются:

1) расчет по среднему тарифу;
2) расчет по переменному тарифу;
3) месячное потребление электроэнергии меньшее, чем 100 квт/ч.;
4) месячное потребление энергии большее или равное 100 квт/ч..

На основе различных комбинаций причин можно перечислить следующие следствия:

101 – минимальная месячная стоимость

102 – процедура А планирования расчета

102 – процедура В планирования расчета.

Шаг 2. Разработка графа причинно-следственных связей. Узлы причин перечисляют по вертикали слева, а узлы следствий – справа (рисунок 3.13).

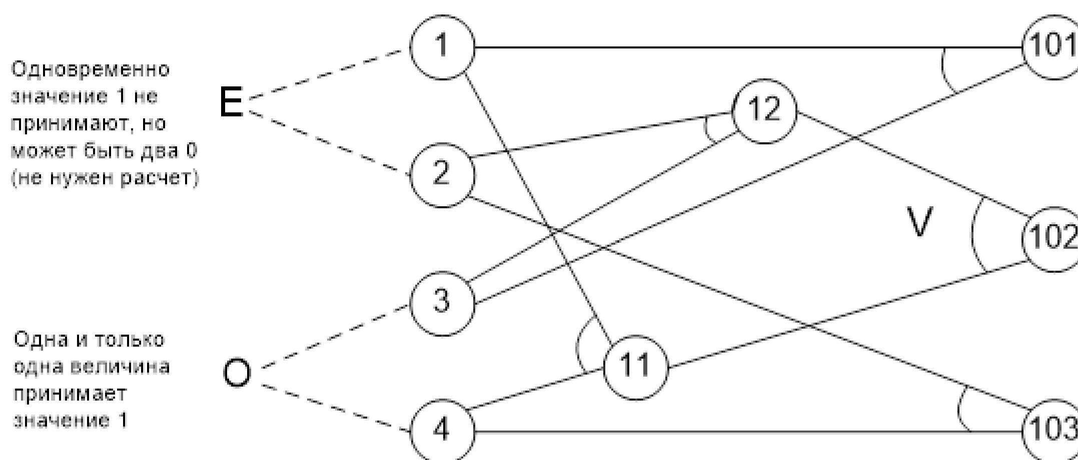

Рисунок 3.13 – Диаграмма причинно-следственных связей

Шаг 3. Генерация таблицы решений. Причины рассматриваются как условия, а следствия – как действия (таблица 3.2).

Порядок генерации:

1 Выбирается некоторое следствие, которое должно быть в состоянии 1.

2 Находятся все комбинации причин (с учетом ограничений), которые устанавливают это следствие в состояние 1. Для этого из следствия прокладывается обратная трасса через граф.

3 Для каждой комбинации причин, приводящих следствие в состояние 1, строится один столбец.

4 Для каждой комбинации причин доопределяются состояния всех других следствий. Они помещаются в тот же столбец таблицы решений.

Действия 1 – 4 повторяются для всех следствий графа.

Таблица 3.2 – Таблица решений

Номера столбцов			1	2	3	4
Условия (причины)	Причины	1	1	0	1	0
		2	0	1	0	1
		3	1	1	0	0
		4	0	0	1	1
	вторичные причины	11	0	0	1	1
		12	0	0	1	0
Действие	следствие	101	1	0	0	0
		102	0	1	1	0
		103	0	0	0	1

Шаг 4. Преобразование каждого столбца таблицы в тестовый вариант. В нашем примере 4 тестовых варианта.

ТВ 1 (столбец 1):

ИД: расчет по среднему тарифу, месячное потребление энергии – 60 квт/ч.

ОЖ.РЕЗ :минимальная месячная стоимость.

ТВ 2 (столбец 2):

ИД: расчет по переменному тарифу, месячное потребление – 90 квт/ч.

ОЖ.РЕЗ : процедура А планирования расчета.

ТВ 3 :

ИД: расчет по среднему тарифу, месячное потребление – 100 квт/ч.

ТВ 4 :

ИД: расчет по переменному тарифу, месячное потребление – 100 квт/ч.

ОЖ.РЕЗ : процедура В планирования расчета.

Предположение об ошибке – метод функционального тестирования программного обеспечения, основанный на опыте и интуиции эксперта-профессионала. Тестировщик с большим опытом и стажем тестирования программ обнаруживает ошибки, подсознательно используя метод предположения об ошибке. Сущность метода заключается в том, чтобы составить список, перечисляющий возможные ошибки и ситуации, в которых ошибки могли проявиться. На основе списка разрабатываются тестовые варианты [8].

4 ИНТЕГРАЦИОННОЕ ТЕСТИРОВАНИЕ

Интеграционное тестирование – тестирование программного обеспечения, на этапе сборки модулей в единый комплекс. Цель интеграционного тестирования – нахождение ошибок взаимодействия модулей (компонент, подсистем). Тестирование выполняется через интерфейс модулей с использованием метода «чёрного ящика» [9, 10].

4.1 Методы интеграционного тестирования

Тестирование интеграции поддерживает сборку цельной программной системы. Тесты проводятся для обнаружения ошибок интерфейса. Перечислим основные категории ошибок интерфейса:

- потеря данных при прохождении через интерфейс;
- отсутствие в модуле необходимой ссылки;
- неблагоприятное влияние одного модуля на другой;
- подфункции при объединении не образуют требуемую главную функцию;
- отдельные (допустимые) неточности при интеграции выходят за допустимый уровень;
- проблемы при работе с глобальными структурами данных.

Существуют два метода сборки модулей:

- монолитный, характеризующийся одновременным объединением модулей в тестируемый комплекс;
- инкрементальный, характеризующийся пошаговым (помодульным) наращиванием комплекса программ с пошаговым тестированием собираемого комплекса [16].

В инкрементальном методе выделяют две стратегии добавления модулей:

- нисходящее тестирование
- восходящее тестирование.

4.1.1 Нисходящее тестирование

При нисходящем тестировании сначала тестируют верхний управляющий модуль программной системы без модулей низкого уровня, вместо которых используют заглушки, затем тестируют модули более низкого уровня.

Заглушка – программный модуль, обладающий тем же интерфейсом, что и замещаемый модуль нижележащего уровня. Заглушка не реализует функциональность замещаемого модуля, а возвращает значения, позволяющие проверить функционирование тестируемого модуля [10].

При данном подходе модули объединяются движением сверху вниз по управляющей иерархии, начиная с главного управляющего модуля. Подчиненные модули добавляются в структуру или в результате поиска в глубину, или в результате поиска в ширину (рисунок 4.1).

Шаги нисходящего тестирования:

1 Главный управляющий модуль (вершина иерархии) используется как тестовый драйвер. Все непосредственно подчиненные ему модули временно замещаются заглушками.

2 Одна из заглушек заменяется реальным модулем. Модуль выбирается поиском в ширину или глубину.

3 После подключения каждого модуля (и установки в нем заглушек) проводится набор тестов, проверяющий полученную структуру.

4 Если в модуле-драйвере уже нет заглушек, производится смена модуля драйвера (поиском в глубину или ширину).

5 Выполняется возврат на шаг 2 (до тех пор, пока не будет построена целая структура).

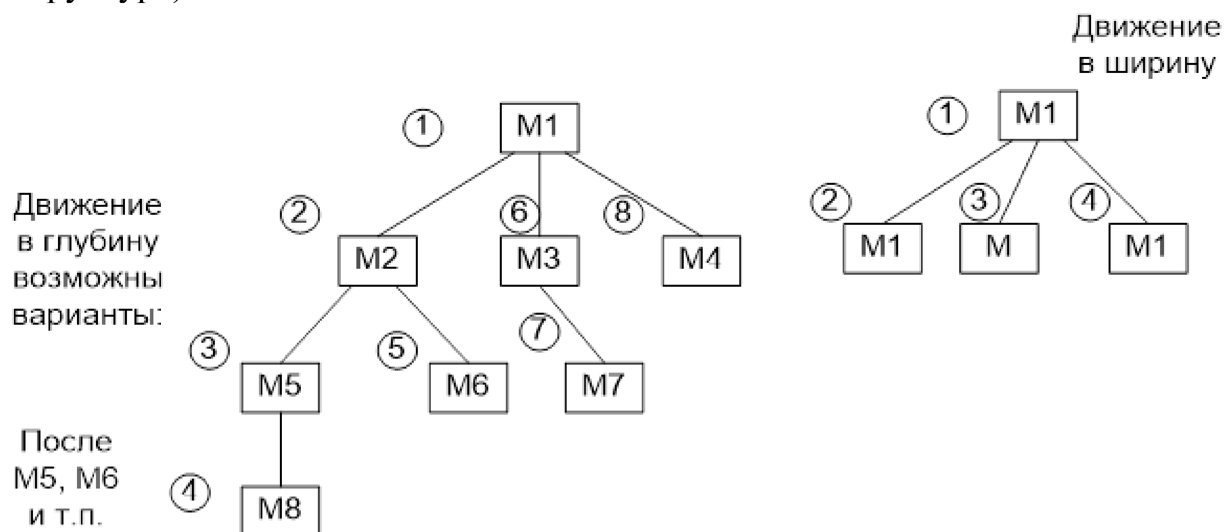

Рисунок 4.1 – Нисходящая интеграция

Достоинство нисходящей интеграции: ошибки в главной, управляющей части системы выявляются в первую очередь.

Недостаток: трудности в ситуациях, когда для полного тестирования на верхних уровнях нужны результаты обработки нижних уровней.

Для борьбы с указанным недостатком существуют три решения:

1) откладывать некоторые тесты до замещения заглушек модулями;
2) разрабатывать заглушки, частично выполняющие функции модулей;
3) подключать модули движением снизу вверх.

Первое решение вызывает сложности в оценке результатов тестирования.

Для реализации второй возможности выбирается одна из следующих категорий заглушек:

- заглушка A – отображает проходящий параметр;
- заглушка B – отображает трассируемое сообщение;
- заглушка C – возвращает величину из таблицы;
- заглушка D – выполняет табличный поиск по ключу (входному параметру) и возвращает связанный с ним выходной параметр (рисунок 4.2).

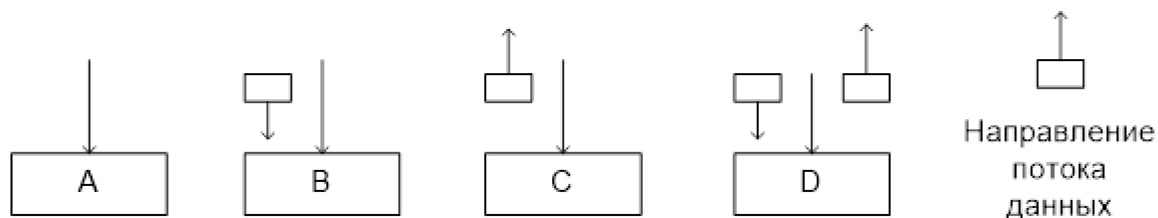

Рисунок 4.2 – Категории заглушек

4.1.2 Восходящее тестирование

При восходящем тестировании сначала тестируются терминальные модули, не зависящие от других модулей, затем тестируются модули, которые зависят от проверенных модулей. Для модуля разрабатывают драйвер.

Драйвер – программный модуль, эмулирующий окружение модуля при тестировании. Драйвер вызывает тестируемый модуль, передает тестовые данные, проверяет результаты работы модуля и корректность реализации его интерфейса. При восходящем тестировании упрощается локализация ошибок.

Шаги методики восходящей интеграции.

1 Модули нижнего уровня объединяются в кластеры (группы, блоки), выполняющие определенную программную функцию.

2 Для координации вводов-выводов тестового варианта пишется драйвер, управляющий тестированием кластеров.

3 Тестируется кластер.

4 Драйверы удаляются, а кластеры объединяются в структуру движением вверх.

Пример восходящей интеграции приведен на рисунке 4.3.

Драйверы могут быть различных типов:

- драйвер А – вызывает подчиненный модуль;
- драйвер В – посылает элемент данных из внутренней таблицы;
- драйвер С – отображает параметр из подчиненного модуля;
- драйвер D – является комбинацией драйверов В и С (рисунок 4.4).

По мере продвижения вверх необходимость в использовании драйверов уменьшается.

4.1.3 Сравнение нисходящего и восходящего тестирования

Нисходящее тестирование:

1) основной недостаток – необходимость заглушек и связанные с ним трудности тестирования;

2) основное достоинство – возможность раннего тестирования главных управляющих функций.

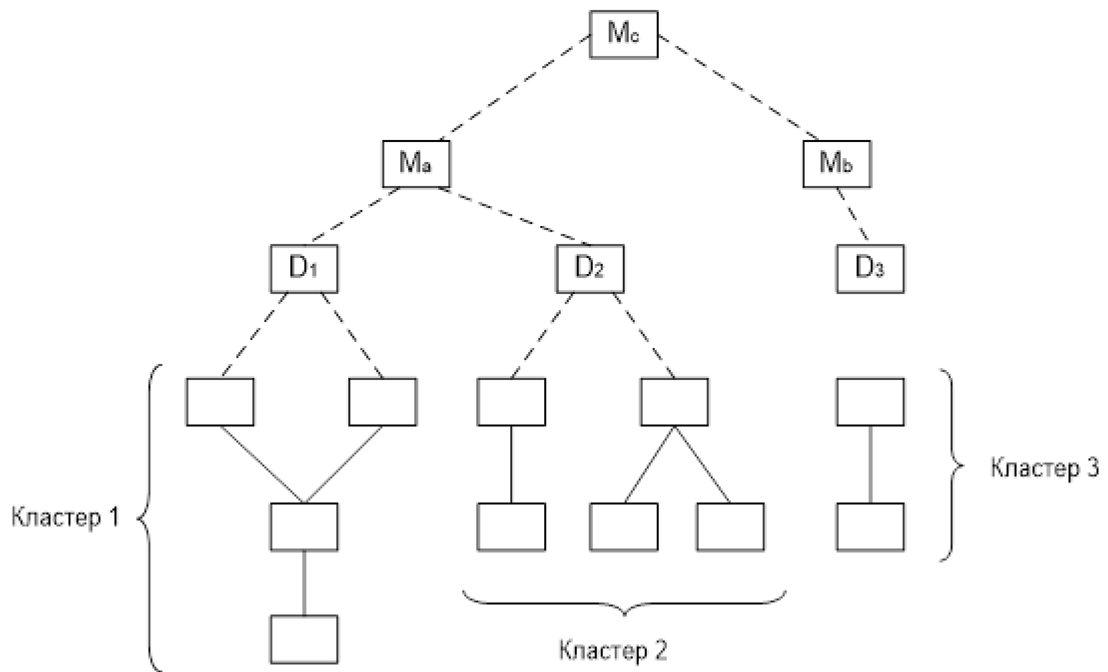

Рисунок 4.3 – Восходящая интеграция

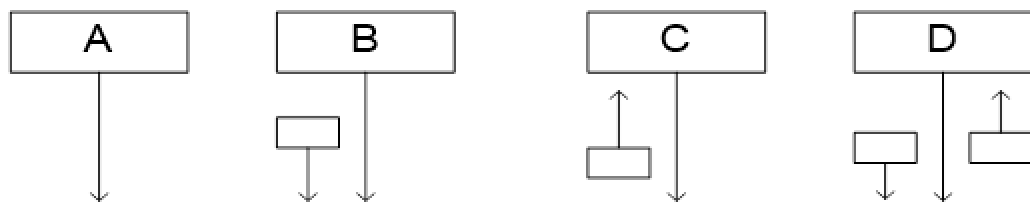

Рисунок 4.4 – Типы драйверов

Восходящее тестирование:

1) основной недостаток – система не существует как объект до тех пор, пока не будет добавлен последний модуль;

2) основное достоинство – упрощается разработка тестовых вариантов, отсутствуют заглушки.

Возможен комбинированный подход, при котором для верхних уровней интеграции применяют нисходящую стратегию, а для нижних уровней – восходящую.

При проведении тестирования интеграции очень важно выявить критические модули. Признаки критического модуля:

1) реализует несколько требований к программной системе;

2) имеет высокий уровень управления (находится достаточно высоко в программной структуре);

3) имеет высокую сложность или склонность к ошибкам (как индикатор может использоваться цикломатическая сложность – ее разумный верхний предел составляет 10);

4) имеет определенные требования к производительности обработки.

Критические модули должны тестироваться как можно раньше. Кроме того, к ним должно применяться регрессионное тестирование (повторение уже выполненных тестов в полном или частичном объеме).

5 КОНТРОЛЬНАЯ РАБОТА

5.1 Назначение, цели и задачи контрольной работы

Разрабатываемое приложение контрольной работы предназначено для формализации решения задачи тестирования программы методом базового пути.

Цель выполнения контрольной работы – закрепление теоретических знаний и повышение практических навыков в разработке программы тестирования методом базового пути на языке С++.

Задачи, решаемые студентом в процессе выполнения контрольной работы:

- изучение теоретического обоснования предметной области;
- разработка алгоритма решения задачи;
- разработка программного приложения;
- документирование контрольной работы.

5.2 Требования к контрольной работе

Контрольная работа включает объектно-ориентированное приложение на языке С++ и пояснительную записку, содержащую разделы:

- введение;
- постановка задачи;
- алгоритм решения задачи;
- результаты работы приложения (скриншоты);
- заключение;
- список использованных источников;
- содержание;

К пояснительной записке прилагается комплект документации:

- спецификация (ГОСТ 19.202-78);
- описание программы (ГОСТ 19.402-78);
- руководство пользователя (ГОСТ 19.505-79);
- руководство программиста (ГОСТ 19.504-79).

Пояснительная записка оформляется в соответствии с требованиями методического указания к оформлению курсовых и дипломных проектов

5.2.1 Требования к функциональным характеристикам

Проектируемое приложение должно обеспечивать выполнение функций:
- ввод исходных данных задачи;
- расчет данных;
- вывод управляющего графа программы;
- расчёт цикломатической сложности управляющего графа программы;
- тестирование и вывод данных;
- оценка тестированности программы;
- расчёт метрик лексического анализа программ (метрики Холстеда, метрики Чепина);
- расчёт метрики структурной сложности программ.

5.2.2 Требования к эксплуатационным характеристикам

- модульность;
- расширяемость.

5.2.3 Требования к программному обеспечению

- среда разработки Microsoft Visual Studio Community 2026;
- язык программирования C++.

5.2.4 Варианты контрольной работы

Разработать приложение на языке C++, формализующее алгоритм метода структурного тестирования базового пути.

Вариант 1

$$y = \begin{cases} a * (x + c)^2 - b, при\ x = 0, \mathrm{b} \neq 0 \\ \dfrac{\text{x-a}}{\text{x-c}}, при\ x = 0\ и\ \mathrm{b} = 0 \\ a + \dfrac{x}{c}, в\ остальных\ случаях \end{cases}$$

Вариант 2

$$y = \begin{cases} a * x^2 - cx + b, при\ x + 10 < 0, \mathrm{b} \neq 0 \\ \dfrac{\text{x-a}}{\text{x-c}}, при\ x + 10 > 0\ и\ b = 0 \\ \dfrac{-x}{a - c}, в\ остальных\ случаях \end{cases}$$

Вариант 3

$$y = \begin{cases} a * x^3 + b * x^2, при\, x < 0, \text{b} \neq 0 \\ \frac{\text{x-a}}{\text{x-c}}, при\, x > 0\; и\; b = 0 \\ \frac{x + 5}{c(x - 10)}, в\; остальных\; случаях \end{cases}$$

Вариант 4

$$y = \begin{cases} a * (x + 7)^2 - b, при\, x < 5, \text{b} \neq 0 \\ \frac{\text{x-c*d}}{\text{a*x}}, при\, x > 5\; и\; b = 0 \\ \frac{x}{c}, в\; остальных\; случаях \end{cases}$$

Вариант 5

$$y = \begin{cases} -\frac{2 * x - c}{c * x - a}, при\, x < 0, \text{b} \neq 0 \\ \frac{\text{x-a}}{\text{x-c}}, при\, x > 0\; и\; b = 0 \\ -\frac{x}{c} + \frac{-b}{2 * c}, в\; остальных\; случаях \end{cases}$$

Вариант 6

$$y = \begin{cases} a * x^3 + b^2 + c, при\, x < 0, \text{b} \neq 0 \\ \frac{\text{x-a}}{c}, при\, x > 0\; и\; c \neq 0 \\ \frac{15 * x}{c}, в\; остальных\; случаях \end{cases}$$

Вариант 7

$$y = \begin{cases} a * x^2 + 2 * b * x + 5, при\, x < 5, \text{b} \neq 0 \\ \frac{\text{3*x-a}}{c}, при\, x \geq 5\; и\; c \neq 0 \\ \frac{10 * x}{c + 6}, в\; остальных\; случаях \end{cases}$$

Вариант 8

$$y = \begin{cases} (a + 2) * x^2 + 5 * b, при\, x < 10, \text{b} \neq 0 \\ \frac{x + a}{\text{7*c}}, при\, x \geq 10\; и\; c \neq 0 \\ \frac{x}{c + 14}, в\; остальных\; случаях \end{cases}$$

Вариант 9

$$y = \begin{cases} a * x^4 + b * x^2, при\, x < 0, \mathrm{b} \neq 0 \\ \dfrac{x + a}{c}, при\, x \geq 0\, и\, c \neq 0 \\ \dfrac{15 * x}{c + 9}, в\ остальных\ случаях \end{cases}$$

Вариант 10

$$y = \begin{cases} a * x^2 + 25 * b * x, при\, x < 0, \mathrm{b} \neq 0 \\ \dfrac{\mathrm{x\text{-}a}}{c}, при\, x \geq 0\, и\, c \neq 0 \\ \dfrac{16 * x}{c + 8}, в\ остальных\ случаях \end{cases}$$

Вариант 11

$$y = \begin{cases} a * x^2 + \dfrac{b}{c}, при\, x < 15, \mathrm{c} \neq 0 \\ \dfrac{\mathrm{x\text{-}a}}{(\mathrm{x\text{-}c})^2}, при\, x > 15\, и\, c = 0 \\ \dfrac{x^2}{c^2}, в\ остальных\ случаях \end{cases}$$

Вариант 12

$$y = \begin{cases} a * x^3 + b^2 + c, при\, x < 0.6, \mathrm{b} + c \neq 0 \\ \dfrac{\mathrm{x\text{-}a}}{\mathrm{x\text{-}c}}, при\, x > 0.6\, и\, \mathrm{b} + c = 0 \\ \dfrac{x}{c} + \dfrac{x}{a}, в\ остальных\ случаях \end{cases}$$

Вариант 13

$$y = \begin{cases} a * x^2 + b, при\, x - 1 < 0, \mathrm{b\text{-}x} \neq 0 \\ \dfrac{\mathrm{x\text{-}a}}{x}, при\, x - 1 > 0\, и\, b + x = 0 \\ \dfrac{x}{c}, в\ остальных\ случаях \end{cases}$$

Вариант 14

$$y = \begin{cases} -a * x^3 - b, при\ x + c < 0, \mathrm{a} \neq 0 \\ \frac{\text{x-a}}{\text{x-c}}, при\ x + c > 0\ и\ a = 0 \\ \frac{x}{c} + \frac{c}{x}, в\ остальных\ случаях \end{cases}$$

Вариант 15

$$y = \begin{cases} -a * x^2 + b, при\ x < 0, \mathrm{b} \neq 0 \\ \frac{x}{\text{x-c}} + 5.5, при\ x > 0\ и\ b = 0 \\ \frac{x}{-c}, в\ остальных\ случаях \end{cases}$$

Вариант 16

$$y = \begin{cases} a * (x + c)^2 - b, при\ x = 0, \mathrm{b} \neq 0 \\ \frac{\text{x-a}}{\text{-c}}, при\ x = 0\ и\ \mathrm{b} = 0 \\ a + \frac{x}{c}, в\ остальных\ случаях \end{cases}$$

Вариант 17

$$y = \begin{cases} a * x^2 - cx + b, при\ x + 10 < 0, \mathrm{b} \neq 0 \\ \frac{\text{x-a}}{\text{x-c}}, при\ x + 10 > 0\ и\ b = 0 \\ \frac{-x}{a - c}, в\ остальных\ случаях \end{cases}$$

Вариант 18

$$y = \begin{cases} a * x^3 + b * x^2, при\ x < 0, \mathrm{b} \neq 0 \\ \frac{\text{x-a}}{\text{x-c}}, при\ x > 0\ и\ b = 0 \\ \frac{x + 5}{c(x - 10)}, в\ остальных\ случаях \end{cases}$$

Вариант 19

$$y = \begin{cases} a * (x+7)^2 - b, \text{при } x < 5, \mathrm{b} \neq 0 \\ \dfrac{\text{x-c*d}}{\text{a*x}}, \text{при } x > 5 \text{ и } b = 0 \\ \dfrac{x}{c}, \text{в остальных случаях} \end{cases}$$

Вариант 20

$$y = \begin{cases} -\dfrac{2 * x - c}{c * x - a}, \text{при } x < 0, \mathrm{b} \neq 0 \\ \dfrac{\text{x-a}}{\text{x-c}}, \text{при } x > 0 \text{ и } b = 0 \\ -\dfrac{x}{c} + \dfrac{-b}{2 * c}, \text{в остальных случаях} \end{cases}$$

Вариант 21

$$y = \begin{cases} a * x^3 + b^2 + c, \text{при } x < 0, \mathrm{b} \neq 0 \\ \dfrac{\text{x-a}}{c}, \text{при } x > 0 \text{ и } c \neq 0 \\ \dfrac{15 * x}{c}, \text{в остальных случаях} \end{cases}$$

Вариант 22

$$y = \begin{cases} a * x^2 + 2 * b * x + 5, \text{при } x < 5, \mathrm{b} \neq 0 \\ \dfrac{\text{3*x-a}}{c}, \text{при } x \geq 5 \text{ и } c \neq 0 \\ \dfrac{10 * x}{c + 6}, \text{в остальных случаях} \end{cases}$$

Вариант 23

$$y = \begin{cases} (a+2) * x^2 + 5 * b, \text{при } x < 10, \mathrm{b} \neq 0 \\ \dfrac{x + a}{\text{7*c}}, \text{при } x \geq 10 \text{ и } c \neq 0 \\ \dfrac{x}{c + 14}, \text{в остальных случаях} \end{cases}$$

Вариант 24

$$y = \begin{cases} a * x^4 + b * x^2, при\ x < 0, \mathrm{b} \neq 0 \\ \dfrac{x + a}{c}, при\ x \geq 0\ и\ c \neq 0 \\ \dfrac{15 * x}{c + 9}, в\ остальных\ случаях \end{cases}$$

Вариант 25

$$y = \begin{cases} a * x^2 + 25 * b * x, при\ x < 0, \mathrm{b} \neq 0 \\ \dfrac{\mathrm{x\text{-}a}}{c}, при\ x \geq 0\ и\ c \neq 0 \\ \dfrac{16 * x}{c + 8}, в\ остальных\ случаях \end{cases}$$

ЗАКЛЮЧЕНИЕ

Информационные технологии являются основным элементом современного общества. Уровень проникновения ЭВМ и интернет в обществе высокий. Информационная система включает аппаратное и программное обеспечение. Программное обеспечение представляет совокупность программ системы обработки информации и программных документов. Программное обеспечение – вид обеспечения вычислительной системы наряду с аппаратным, математическим, информационным, лингвистическим, организационным и методическим обеспечением. Программное обеспечение реализует возможности вычислительного комплекса и взаимодействует с пользователем, поэтому к программному обеспечению предъявляются жёсткие требования в отношении качества. На современном этапе развития информационных систем программное обеспечение становится более сложным и ценным. Стоимость программного обеспечения достигает 30-90% стоимости информационных систем. Возрастающая сложность программного обеспечения приводит к увеличению количества ошибок в программном коде, что отрицательно сказывается на результатах работы программного обеспечения. Обеспечение качества программного продукта является актуальной задачей на современном этапе развития информационных технологий.

Тестирование – обнаружение ошибок в программном приложении.

Выполнение контрольной работы позволит студентам закрепить теоретические знания по дисциплине «Управление качеством и тестирование программного обеспечения» и приобрести практические навыки в разработке программных приложений.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1 Семахин А. М. Методы верификации и оценки качества программного обеспечения : учебное пособие / А. М. Семахин. – Курган : Изд-во КГУ, 2018. 150 с.

2 Благодатских В. А. Стандартизация разработки программных средств: учебное пособие / В. А. Благодатских, В. А. Волнин, К. Ф. Поскакалов, под. ред. О. С. Разумова. – Москва : Финансы и статистика, 2005. – 288 с.

3 Черников Б. В. Управление качеством программного обеспечения / Б. В.Черников. – Москва : ИД «ФОРУМ»: ИНФРА-М. 2012. – 240 с.

4 Решка Джеф. Тестирование программного обеспечения. Внедрение, управление и автоматизация / Джеф Решка, Джон Пол, Элфрид Дастин. – Москва: Лори, 2012. – 600 с.

5 Сандлер Кори. Искусство тестирования программ / Кори Сандлер, Том Баджет, Гленфорд Майерс. – Москва : Вильямс, 2012. – 272 с.

6 Сэм Канер. Тестирование программного обеспечения / Канер Сэм, Фолк Джек. – Москва : ДиаСофт, 2001. – 538 с.

7 Котляров В. П. Основы тестирования программного обеспечения / В.П. Котляров. – Москва : Интернет Университет Информационных Технологий; БИНОМ. Лаборатория знаний, 2006. – 285 с.

8 Майерс Г. Дж. Искусство тестирования программ / Г. Дж. Майерс. – Москва : Финансы и статистика, 1982. – 176 с.

9 Майерс Г. Дж. Надежность программного обеспечения / Г. Дж. Майерс. – Москва : Мир, 1980. – 360 с.

10 Анашкина Н. В. Технологии и методы программирования / Н. В. Анашкина. – Москва : Издательский центр «Академия», 2012. – 380 с.

11 Макконнелл С. Совершенный код / С. Макконелл. – Санкт-Петербург : «Питер», 2005. – 896 с.

12 Бейзер Б. Тестирование черного ящика / Б. Бейзер. – Санкт-Петербург : «Питер», 2005. – 318 с.

13 Брауде Э. Технология разработки программного обеспечения / Э. Брауде. – Санкт-Петербург : «Питер», 2004. – 655 с.

Семахин Андрей Михайлович

УПРАВЛЕНИЕ КАЧЕСТВОМ И ТЕСТИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Методические указания
к выполнению контрольных работ
для студентов направлений 09.03.03, 09.03.04

Редактор В. А. Лисина

Подписано в печать 28.04.26 Формат 60х84 1/16 Бумага 80 г/м2
Печать цифровая Усл. печ. л. 2,25 Уч.-изд. л. 2,25
Заказ 23 Тираж 25 Не для продажи

БИЦ Курганского государственного университета.
640020, г. Курган, ул. Советская, 63/4.
Курганский государственный университет.