11 Понятие и классификация исключительных ситуаций.

Если отладка - уничтожение багов, то программирование - их создание

11.1. Понятие и методы тестирования

Тестирование программы является одной из составных частей более общего понятия – «отладка программы». Под **отладкой** понимается процесс, позволяющий получить программу, функционирующую с требуемыми характеристиками в заданной области изменения входных данных.

Процесс отладки включает:

1. действия, направленные на выявление ошибок (тестирование);
2. диагностику и локализацию ошибок (определение характера и местоположения ошибок);
3. внесения исправлений в программу с целью устранения ошибок.

Из 3-х перечисленных видов работ самым трудоемким и дорогим является тестирование, затраты на которое для типичных программных изделий (ПИ) приближаются к 40% общих затрат на разработку.

Программы, как объекты тестирования, имеют ряд особенностей, которые отличают процесс их тестирования от общепринятого, применяемого при разработке аппаратуры и других технических изделий. Особенностями ПИ являются:

1. отсутствие эталона (программы), которой должна соответствовать тестируемая программа;
2. высокая сложность программ и принципиальная невозможность исчерпывающего тестирования;

3. практическая невозможность создания единой методики тестирования (формализации процесса тестирования) в силу большого разнообразия ПИ по их сложности, функциональному назначению, области использования и т.д.

Применительно к ПИ **тестирование** – это процесс многократного выполнения программы с целью обнаружения ошибок. **Цель тестирования** – выявить как можно большее число ошибок. Принципы тестирования:

1. процесс тестирования более эффективен, если проводится не автором программы;
2. тестовый набор должен включать 2 компонента: описание входных данных и описание точного и корректного результата, соответствующего набору входных данных;
3. Тесты для неправильных и непредусмотренных входных данных должны разрабатываться также тщательно, как и для правильных, предусмотренных. Тестовые наборы из области недопустимых входных значений обладают большей обнаруживающей способностью, чем тесты, соответствующие корректным входным данным;
4. Необходимо проверять не только, делает ли программа то, для чего она предназначена, но и не делает ли она то, что не должна делать;
5. Вероятность наличия необнаруженных ошибок в части программы пропорциональна числу ошибок, уже обнаруженных в этой части.

Методы тестирования:

Статическое тестирование – наиболее формализованное, базируется на правилах структурного построения программы и обработки данных. Проверка выполняется путем формального анализа текста программы на языке программирования. Операторы и операнды текста программы в символьном виде (символьное тестирование).

Детерминированное тестирование – наиболее трудоемкий и детализированный метод, который требует многократного выполнения программы на компьютере с использованием определенных, специальным образом подобранных тестовых наборов данных. При использовании метода контролируется каждая комбинация исходных данных и соответствующие результаты, а также каждое утверждение в спецификации тестируемой программы.

Стохастическое тестирование – применяется для комплексного тестирования ПИ, когда невозможно из-за сложности ПИ, перебрать все комбинации исходных данных и проконтролировать результаты функционирования каждого из них. Данный метод предполагает использование в качестве исходных данных множество случайных величин с соответствующими распределениями, а для сравнения полученных результатов используется также распределение случайных величин. Применяется в основном для обнаружения ошибок, а для диагностики и локализации переходят к детерминированному тестированию с использованием конкретных значений исходных данных из области изменения ранее используемых случайных величин (генератор случайных чисел).

Тестирование в реальном масштабе времени применяется к ПИ, которые предназначены для работы в системах реального времени. В процессе этого тестирования проверяются результаты обработки исходных данных с учетом времени их поступления, длительности и приоритетности обработки, динамики использования памяти и взаимодействия с другими программами. При обнаружении отклонений результатов выполнения программы от ожидаемых для локализации ошибок приходится фиксировать время и переходить к детерминированному тестированию.

11.2. Методы проектирования тестовых наборов данных (для детерминированного тестирования)

Детерминированное тестирование основывается на двух подходах:

Структурное тестирование, тестирование программы как «белого ящика» (стратегия тестирования, управляющего логикой программы), предполагает детальное изучение текста (логики) программы и построение таких входных наборов данных, которые позволили бы при многократном выполнении программы обеспечить выполнение максимально возможного количества маршрутов, логических ветвлений, циклов и т.д.

Функциональное тестирование, или тестирование программы как «черного ящика» (тестирование по входу – выходу), полностью абстрагируется от логики программы, предполагается, что программа – «черный ящик», а тестовые наборы выбираются на основании анализа входных функциональных спецификаций.

Понятие эффективного тестового набора данных связано с невозможностью полного тестирования программы. Т.к. тестируемые программы состоят из множества сложных участков, то исчерпывающее тестирование маршрутов не только невыполнимо, но и невозможно.

Подмножество всех возможных тестов, которое имеет максимальную вероятность обнаружения большинства ошибок, называется **эффективным.**

При построении тестовых наборов данных по принципу «белого ящика» руководствуются следующими критериями:

Покрытие операторов. Этот критерий предполагает выбор такого тестового набора данных, который вызывает выполнение каждого оператора в программе хотя бы 1 раз (критерий очень слабый).

Покрытие узлов ветвления (покрытие решений). Предполагает разработку такого количества тестов, чтобы в каждом узле ветвления был обеспечен переход по веткам «истина» и «ложь» хотя бы 1 раз.

Покрытие условий. Если узел ветвления содержит более 1 условия, тогда нужно разрабатывать число тестов, достаточное для того, чтобы возможные результаты каждого условия в решении выполнялись, по крайней мере 1 раз. (case … of)

Комбинаторное покрытие условий. Для отслеживания таких ошибок используют комбинаторное покрытие условий. Этот критерий требует создания такого числа тестов, чтобы все возможные комбинации результатов условия в каждом решении и все точки входа выполнялись по крайней мере 1 раз.

При построении тестов по стратегии «черного ящика» программа рассматривается как «черный ящик» (не известны текст и ее логика), а исходной информацией для тестовых наборов служит их описание. К этой стратегии относятся методы:

Метод эквивалентного разбиения. Построение тестов этим методом осуществляется в 2 этапа:

- выделение классов эквивалентности;
- построение тестов.

Классом эквивалентности называют множество входных значений, каждое из которых имеет одинаковую вероятность обнаружения конкретного типа ошибок.

Классы эквивалентности выделяются путем анализа входного условия и разбиением его на 2 или более группы. Для каждого условия существует правильный и неправильный класс эквивалентности.

При выделении классов эквивалентности целесообразно придерживаться следующих правил.

1. Если входное условие описывает область значений, то определяется 1 правильный класс и 2 неправильных. Например, имя файла в MS Dos должно содержать 8 символов. Правильный класс эквивалентности, когда имя от 1 до 8 символов, неправильные классы: имя пустое и имя больше 8 символов;

2. Если входное условие описывает конечное число конкретных значений, то определяется правильный класс для каждого значения и 1 неправильный. Например, для записи чисел в двоичной системе счисления используются 0 и 1. Правильные классы эквивалентности: 0 и 1, неправильный класс эквивалентности - 5;
3. Если входное условие описывает ситуацию «должно быть», то определяется 1 правильный класс и 1 неправильный. Например, имя переменной должно начинаться с буквы. Правильный класс эквивалентности – первый символ буква, неправильный – цифра.

Полученная информация сводится в таблицу 11.1:

Таблица 11.1.

Таблица классов эквивалентности

Входные условия	Правильные классы эквивалентности	Неправильные классы эквивалентности
…	…	…

На основе классов эквивалентности строятся тестовые наборы. Причем для правильных классов нужно стремиться к минимальному числу тестовых наборов, т.е. каждый тест должен покрывать во возможности большее число правильных классов эквивалентности.

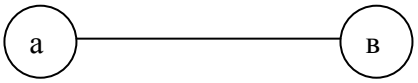
Для каждого неправильного класса строится хотя бы 1 тестовый набор.

Метод функциональных диаграмм. Метод заключается в преобразовании входных описаний программы в функциональную диаграмму (диаграмму причинно – следственных связей) с помощью простейших булевских отношений, построения таблицы решений (методом обратной трассировки), которая является основой для написания эффективных тестовых наборов данных. Последовательность следующая:

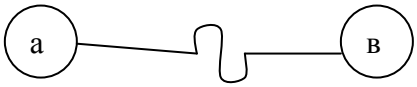
1. В описании программы выделяют причины и следствия. **Причины** – это отдельное входное условие. **Следствие** – выходное условие или результат преобразования системы. Каждой причине и следствию присваивают уникальный номер.

2. Анализируют семантическое содержание определения, которое преобразуется в булевский граф, связывающий причины и следствия. Каждая вершина графа может находиться в состоянии «истина» или «ложь».

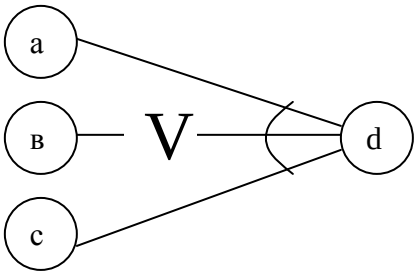
Тождество, если а=1, то в=1, если а=0, то в=0.



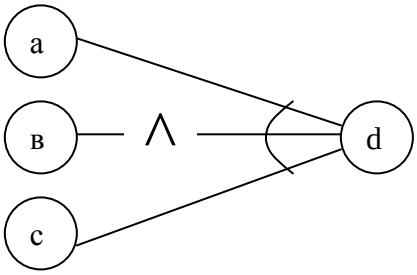
Не (отрицание), если а=1, то в=0, если а=0, то в=1.



Или: если а=1 или в=1 или с=1, то d=1. в противном случае d=0.



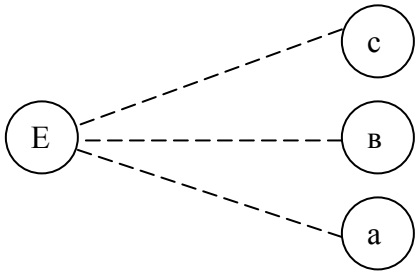
И: если а=1 и в=1 и с=1, то d=1, в противном случае d=0.



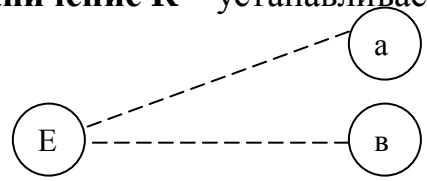
3. Диаграмма дополняется примечаниями, задающими ограничения и описывающими комбинации причин и (или) следствий, которые являются невозможными из-за синтаксических или внешних ограничений.

Используются следующие обозначения:

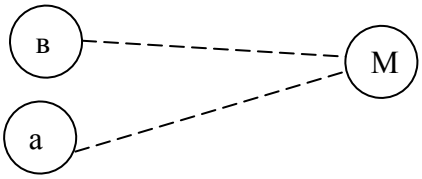
Ограничение Е - устанавливает, что а, в и с не могут принимать значение истина одновременно.



Ограничение R – устанавливает, что если а=1, то и в=1.



Ограничение М является ограничением для следствий и устанавливает, что если следствие а истинно, то следствие в должно быть ложно.



4. По полученной функциональной диаграмме строится таблица решений. Для этого каждое следствие по очереди устанавливается в истину и прослеживается обратный путь ко всем причинам, связанным с этим следствием, и фиксируется их состояние. Каждый столбец таблицы соответствует тесту.

5. Столбцы решений преобразуются в тесты.

Рассмотрим пример. Методом функциональных диаграмм составить тестовый набор для программы, которая осуществляет контроль записей файла, содержащего информацию о студентах института заочной формы обучения. 1-я позиция имени файла – буква «З» (заочная форма обучения), 2-я позиция имени файла зависит от специальности. Рассмотрим только две специальности: маркетинг (обозначается буквой «М») и финансы (обозначается буквой «Ф»). Мы должны получить файлы с именами ЗМ и ЗФ. В случае ошибки в первой позиции выдается сообщение «ошибка в

первой позиции имени файла», ошибка во второй позиции файла вызывает сообщение «Ошибка во второй позиции имени файла».

1. Выделяем причины и следствия и присваиваем им номера.

Причины:

1 – буква «З» в первой позиции;

2 – буква «М» во второй позиции;

3 – буква «Ф» в третьей позиции.

Следствия:

20 – сформирован файл ЗМ;

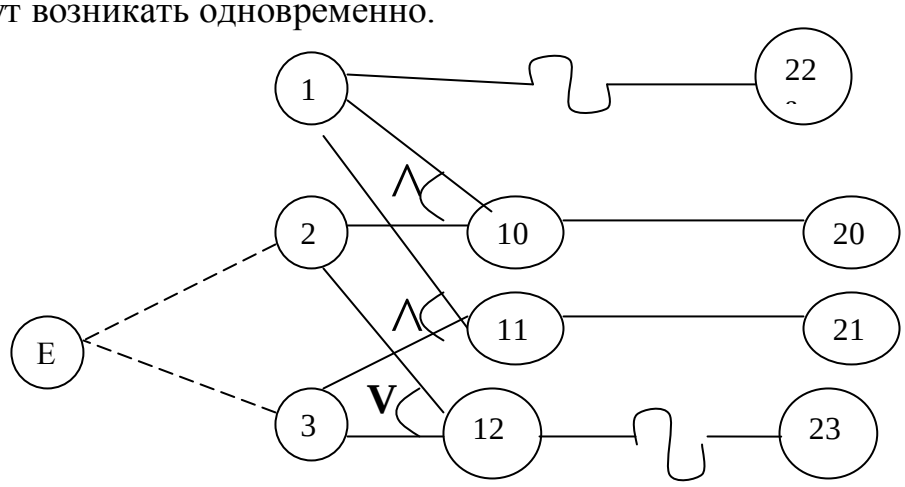
21 – сформирован файл ЗФ;

22 – сообщение «Ошибка в первой позиции имени файла»

23 – сообщение «Ошибка во второй позиции имени файла».

2. Строится функциональная диаграмма, связывающая эти причины и следствия. Для наглядности графа при построении используются промежуточные вершины: 10, 11, 12.

3. Построенный граф снабжается ограничениями. Причины 2 и 3 не могут возникать одновременно.



4. Строится таблица решений по графу (таблица 11.2). Для этого фиксируем в состоянии истина поочередно все следствия. Например, следствия 22 может быть в состоянии истина только в том случае, если причина 1, с которой она связана, находится в позиции ложь. Аналогично обрабатываются остальные причины.

Таблица 11.2.

Таблица решений

Причины	1	0	1	1	-
	2	-	1	-	0
	3	-	-	1	0
Следствия	22	1	-	-	-
	20	-	1	-	-
	21	-	-	1	-
	23	-	-	-	1

5.Переходим от таблицы решений к тестовым наборам. Каждый столбец таблицы решений соответствует тестовому набору. «1» – означает выполнение причины, «0» – означает не выполнение, «-» - означает, что эта причина не влияет на значение следствия.

Получим тестовые наборы:

1. АМ
2. ЗМ
3. ЗФ
4. ЗА

Каждый из рассмотренных методов обеспечивает создание определенного набора тестов, но ни один из них сам по себе не может дать исчерпывающий набор тестов. Поэтому при разработке тестовых наборов следует придерживаться разумного сочетания всех рассмотренных методов.

11.3. Сборка программ при тестировании.

Разработка больших, многомодульных программных комплексов требует использования специальных способов сборки их при тестировании.

Прежде чем приступить к тестированию программного комплекса в целом, нужно, чтобы составляющие его части (отдельные модули или набор модулей) были тщательно оттестированы. Это необходимое условие получения надежных программных комплексов возможно обеспечить на уровне тестирования отдельных модулей, т.к. только на этом уровне возможно управление комбинаторикой тестирования: допускается

одновременное тестирование нескольких модулей, контроль за передачей параметров в модули и т.д.

Классификация методов сборки модулей.

Выделяют два основных вида сборки программ при тестировании:

1. Монолитный;
2. пошаговый:
   - восходящая (снизу – вверх);
   - нисходящая (сверху – вниз).

Монолитный метод сборки предполагает выполнение автономного тестирования каждого модуля, а затем их одновременную сборку и тестирование в комплексе.

Пошаговое тестирование предполагает последовательное подключение к набору уже оттестированных модулей очередного тестируемого модуля.

Пример. Рассмотрим программу, состоящую из 9 модулей (рис. 11.1).

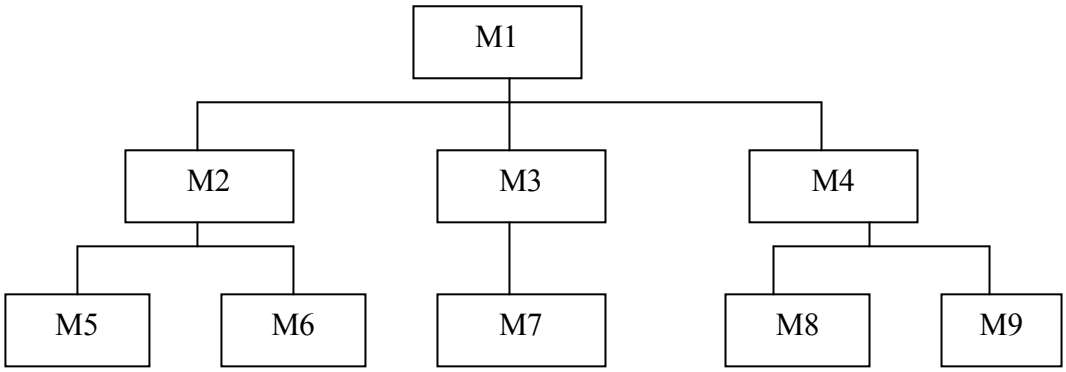


Рис. 11.1. Структура программы

При монолитном тестировании все 9 модулей, входящих в программу, тестируются независимо друг от друга, последовательно или параллельно. Затем они собираются в программу. Для автономного тестирования каждого модуля нужны модуль – драйвер (отслеживающий модуль) 1 или несколько модулей – заглушек (имитатор). Для рассматриваемого примера модули – драйверы нужны для всех модулей кроме M1, а модули – заглушки нужны для всех модулей кроме M5, M6, M7, M8, M9 (т.е. модулей самого низшего уровня). Таким образом, при монолитной сборке для автономного

тестирования составляющих программный комплекс модулей дополнительно необходимо разработать 8 модулей – драйверов и 8 модулей - заглушек.

Драйвер – это модуль, обеспечивающий вызов и передачу параметров модулю необходимых входных данных и обработку результатов.

Заглушка – это модуль, имитирующий работы модулей, вызываемых тестируемым модулем.

Метод пошаговой сборки предполагает, что модули тестируются не автономно, а подключают поочередно для выполнения теста к набору уже ранее оттестированных модулей. При таком подходе возможны два варианта: сверху – вниз или снизу – вверх.

Пусть тестируем сверху – вниз, тогда:

Для М1 нужны 3 заглушки;

Далее подключаем реальный модуль М2, для которого разрабатываются 2 заглушки, и тестируем М1-М2. Затем заглушка М5 заменяется реальным модулем М5 и тестируется цепочка М1 – М2 – М5. Процесс продолжается до тех пор, пока не будет протестирован весь комплекс.

В рассматриваемом примере есть возможность некоторого распараллеливания работ и автономного тестирования цепочек:

М1- М2 – М5 (М6)

М1 – М3 – М7

М1 – М4 – М8 (М9)

Видно, что при пошаговой сборке сверху – вниз нужно разработать 8 заглушек, но не нужны драйверы.

При тестировании снизу – вверх процесс идет в обратном направлении: тестируются модули низшего уровня: М5, М6, М7, М8, М9. Для каждого из них нужен драйвер. Далее параллельно можно проводить тестировании М5 – М2, М6 – М2, М7 – М3, М8 – М4, М9 – М4. Затем подключать М1 и провести комплексное тестирование всей программы. Таким образом, при восходящем тестировании нужно разработать 8 драйверов, но не нужны заглушки.

Сравнение обоих методов сборки записаны в таблице 11.3.

Таблица 11.3.

Сравнение монолитной и пошаговой сборки.

Метод	Достоинства	Недостатки
Монолитная	1. возможность ведения параллельного тестирования, что приводит к сокращению времени тестирования.	1. требует больших затрат, т.к. предполагает дополнительную разработку драйверов и заглушек; 2. модули «не видят друг друга» до последней фазы, что может привести к трудности определения ошибок при передаче параметров и в интерфейсах.
Пошаговая	1. разрабатывают либо только заглушки (сверху – вниз), либо драйверы (снизу – вверх), что сокращает материальные затраты. 2. раньше обнаруживаются ошибки в интерфейсах модулей и в передаче параметров между модулями, т.к. раньше начинается сборка программы.	1. трудно вести параллельное тестирование модулей, что удлиняет время тестирования.

Выбор способа сборки зависит от конкретных условий разработки и особенностей тестируемого программного комплекса.

11.4. Классификация исключительных ситуаций

Ошибки, возникающие в ходе разработки и эксплуатации программ, могут быть следующих видов: синтаксические, логические, динамические.

Синтаксические ошибки возникают при нарушении синтаксиса языка. Они выявляются и устраняются при компиляции программы.

Логические (семантические) ошибки являются следствием неправильного алгоритма и проявляются при выполнении программы. Определяются на этапе тестирования.

Динамические ошибки возникают при выполнении программы и являются следствием неправильной работы операторов, процедур, функций или методов программы, а также операционной системы. Динамические ошибки называются также ошибками времени выполнения (Runtime errors). Например, деление на ноль в операторе присваивания.

Программист должен предвидеть возможность возникновения динамической ошибки и программировать ее обработку. В Delphi для обработки таких ошибок введено понятие исключительной ситуации.

Исключительная ситуация представляет собой нарушение условий выполнения программы, вызывающее прерывание или полное прекращение ее работы. Обработка исключительных ситуаций состоит в нейтрализации динамической ошибки, вызвавшей ее.

Исключительные ситуации могут возникать по различным причинам, например, в случае нехватки памяти, из-за ошибки преобразования, в результате выполнения вычислений и др. В любом случае независимо от источника ошибки приложение получает информацию о ее возникновении. Возникшая исключительная ситуация остается актуальной до тех пор, пока не будет обработана глобальным обработчиком или локальными процедурами.

В Delphi для обработки динамических ошибок в выполняемый файл приложения встраиваются специальные фрагменты кода, предназначенные для реагирования на исключительные ситуации. В результате возникающие во время выполнения программы динамические ошибки автоматически преобразовываются в соответствующие объекты исключительных ситуаций. Объект исключения содержит информацию о типе ошибки и при возникновении исключительной ситуации заставляет программу или ее поток

временно приостанавливаться. Объекты исключений автоматически разрушаются после обработки исключительной ситуации.

Для обработки исключительных ситуаций приложение имеет один глобальный обработчик и несколько специализированных процедур – обработчиков, реагирующих на соответствующие исключения. Каждую исключительную ситуацию обрабатывает свой специализированный локальный обработчик. Исключение, не имеющее своего локального обработчика, обрабатывается глобальным обработчиком приложения.

Механизм глобальной обработки исключений реализуется через объект **application**, который имеет любое приложение. Его можно просмотреть в файле проекта (*.dpr).

11.5. Локальная обработка исключений.

Для ее реализации существуют две конструкции **try … finally** и **try … except.** Обе конструкции имеют похожий синтаксис, но различное назначение. Блоки try включают операторы программы, которые могут вызвать исключительную ситуацию, например, запись на диск или преобразование данных.

Выбор конструкции зависит от операторов программы и действий, выполняемых при возникновении ошибки. Конструкции try могут содержать один или более операторов, а также быть вложенными друг в друга.

```
Try
// Операторы, выполнение которых может вызвать ошибку
finally
// Операторы, которые должны быть выполнены даже  в случае ошибки
end;
```

Эта конструкция применяется для выполнения всех необходимых действий перед передачей управления на следующий уровень обработки

ошибки или глобальному обработчику. Например, освобождение оперативной памяти или закрытие файла. Эта конструкция не обрабатывает объект исключительной ситуации и не удаляет его, а выполняет операторы, которые должны выполниться даже в случае возникновения ошибки.

Конструкция работает следующим образом: если в любом из операторов секции try возникла исключительная ситуация, то управление передаются первому оператору секции finally для выполнения всех операторов секции. Если исключительная ситуация не возникла, то последовательно выполняются все операторы обеих секций. Например, обработка исключительной ситуации деления на ноль.

```
Procedure tform1.button1click(sender:tobject);
Var
A: integer;
C,x: real;
Begin
 A:=0; c:=0; x:=0;
 Try
        A:=strtoint(edit1.text);
        C:=a/strtofloat(combobox1.text);
        X:=sqrt( c);
Finally
 Label1.caption:='Ответ a='+inttostr(a)+#13+'c='+floattostr(c)+#13+'x='
+floattostr(x);
End;
```

В метку выводятся данные независимо от того, произошла ошибка при делении или нет. Затем выполнение программы будет прервано и в случае ошибке выдана диагностика.

Вторая конструкция:

```
Try
// Операторы, выполнение которых может вызвать ошибку
except
// Операторы, которые должны быть выполнены в случае ошибки
end;
```

Эта конструкция, в отличие от предыдущей, применяется для перехвата исключительной ситуации и предоставляет возможность ее обработки. Если

в секции try возникает исключительная ситуация, то управление передается первому оператору секции except. Если исключительная ситуация не возникла, то операторы секции except не выполняются. В случае проявления ошибки операторы секции except могут ликвидировать ошибочную ситуацию и восстановить работоспособность программы. Для исключений, обрабатываемых в конструкции try … except, глобальный обработчик не вызывается и обработку ошибок обеспечивает программист. Например, ввод числового значения в поле редактирования.

```
Procedure tform1.button1click(sender:tobject);
Begin
Try
 Edit1.text:=inttostr(strtoint(edit1.text)+1);
Except
     Messagedlg('содержимое поля edit равно' +edit1.text+#10#13+'Это не число!',
      mterror,[mbok],0);
     edit1.text:='';
     if edit1.canfocus then edit1.setfocus;
end;
end;
```

При нажатии на кнопку число, отображаемое в поле редактирования, увеличивается на единицу. В случае неправильного ввода данных может возникнуть ошибка преобразования. При возникновении исключительной ситуации выдается предупреждение, сбрасывается введенная информация и устанавливается фокус ввода на поле редактирования для повторного набора числа. Если преобразование и увеличение прошли корректно, то операторы секции except не выполняются.

Секция except может быть разбита на несколько частей конструкциями **on … do**, позволяющими анализировать класс исключительной ситуации с целью ее обработки. Конструкция **on …do** применяется в случаях, когда

действия по обработке исключительной ситуации зависят от класса исключения:

On {идентификатор: класс исключения} do

{ оператор обработки исключения этого класса};

else {оператор};

При обработке конструкции on класс возникшей исключительной ситуации сравнивается с указанным классом исключения. В случае совпадения классов выполняется оператор после слова do, реализующий обработку данной исключительной ситуации.

Идентификатор (произвольное имя, заданное программистом) является необязательным элементом и может отсутствовать, при этом не ставится и разделительный знак «:». Идентификатор – это локальная переменная, представляющая собой экземпляр класса исключения, который можно использовать для доступа к объекту возникшего исключения. Эта переменная доступна только внутри «своей» конструкции on… do.

Если класс возникшей исключительной ситуации не совпадает с проверяемым классом, то выполняется оператор после слова else. Элемент else и последующий оператор необязательны.

Если в секции except несколько конструкций on…do, то элемент else должен быть расположен в конце секции, чтобы он относился ко всей совокупности конструкций on…do. Следующий после else оператор выполняется в том случае, если исключительная ситуация не обработана ни одним из операторов, расположенным в любой из конструкций do секции. Операторы, следующие за словами do и else, могут быть составными. Принцип работы аналогичен case of.

```
Procedure tform1.button1click(Sender:Tobject);
Var
  X,y,res: real;
Try
  X:=strtoint(edit1.text);
```

```
    Y:=strtoint(edit2.text);
     Res:=x/y;
    Edit3.text:=floattostr(res);
  Except
    On Ezerodivide do begin
       Messagedlg('Попытка деления на ноль!', mterror,[mbok],0);
        Edit2.setfocus;
        Edit3.text:='ошибка!';
      End;
     On Eo: Econverterror do begin
        Messagedlg('Ошибка преобразования!'+#10#13+eo.message,
mterror,[mbok],0);
        Edit1.setfocus;
        Edit3.text:='ошибка!';
     End;
    Else begin
       messagedlg('Ошибка не идентифицирована',  mtwarning,
  [mbok],0);
        Edit1.setfocus;
        Edit3.text:='ошибка!';
     End;
  End;
```

В первом и втором полях редактирования содержатся два целых числа. При нажатии кнопки происходит деление первого числа на второе и результат помещается в третье поле редактирование. Выполняя эти действия операторы могут вызвать ошибку, поэтому они помещены в секцию try. Если возникает исключительная ситуация, она анализируется в секции except. Проверяются следующие варианты: **Econverterror** (ошибка преобразования данных) и **Ezerodivide** (деление на ноль). При этом используются две конструкции on…do. Эти ошибки распознаются и выдается соответствующее

сообщение. Любая другая исключительная ситуация не будет распознана, в этом случае выполняется составной оператор, следующий за else.

Во втором случае использовалось имя переменной **eo**, для ссылки на объект исключения, например, для доступа к сообщению о характере ошибки (Eo.message).

Конструкции try могут быть вложенными и размещаться одна в другой. При этом и внешняя и внутренняя конструкции могут быть любыми из двух рассмотренных видов. Обязательное условие – полная вложенность (аналогично циклам).

```
Try
  {операторы}
   try
      {операторы}
    finally
          {операторы}
     end;
except
      {операторы}
end;
```

или

```
try
    {операторы}
   try
         {операторы}
    except
        {операторы}
     end;
finally
   {операторы}
end;
```

В случае, когда какие-либо действия должны быть выполнены независимо от того, произошла ошибка или нет, удобно использовать конструкцию try…finally. Однако, эта конструкция не обрабатывает исключительную ситуацию, а только информирует о ней и смягчает ее последствия. Если требуется выполнить и локальную обработку

исключительной ситуации, можно включить блок try…finally в конструкцию try…except. При возникновении исключения это позволит выполнить обязательные операторы секции finally и обработать исключение операторами секции except. В таблице 11.4. представлены стандартные константы для обработки исключений.

Таблица 11.4.

Стандартные константы обработки исключений.

Константа	Значение
Eabort	«тихое» исключение, используется для прерывания текущего модуля.
Eoutofmemory	Не достаточно оперативной памяти для выполнения операции
Econverterror	Ошибка преобразования типов данных
Ezerodivide	Деление на ноль
Einouterror	Ошибка ввода/вывода любого файла
Efcreateerror	Ошибка создания файла
efopenerror	Ошибка открытия файла.

Контрольные вопросы

1. Понятие и этапы отладки программ. Общие принципы и методы тестирования программ.
2. Понятие эффективного тестового набора. Методы детерминированного тестирования.
3. Методы сборки программ при тестировании. Понятие программы - драйвера и программы – заглушки.
4. Понятие исключительной ситуации. Методы ее обработки.
5. Способы локальной обработки исключительных ситуаций.