## Основные понятия и принципы тестирования ПО

Тестирование — процесс выполнения программы с целью обнаружения ошибок. Шаги процесса задаются тестами.

Каждый тест определяет:

* свой набор исходных данных и условий для запуска программы;
* набор ожидаемых результатов работы программы.

Другое название теста — тестовый вариант. Полную проверку программы гарантирует *исчерпывающее тестирование.* Оно требует проверить все наборы исходных данных, все варианты их обработки и включает большое количество тестовых вариантов. Увы, но исчерпывающее тестирование во многих случаях остается только мечтой — срабатывают ресурсные ограничения (прежде всего, ограничения по времени).

Хорошим считают тестовый вариант с высокой вероятностью обнаружения еще не раскрытой ошибки. Успешным называют тест, который обнаруживает до сих пор не раскрытую ошибку.

Целью проектирования тестовых вариантов является систематическое обнаружение различных классов ошибок при минимальных затратах времени и стоимости.

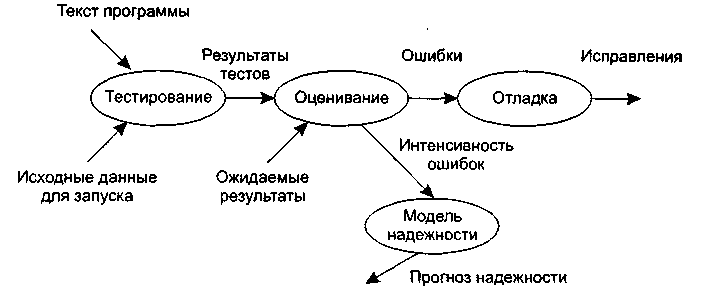
Важен ответ на вопрос: что может тестирование?

Тестирование обеспечивает:

* обнаружение ошибок;
* демонстрацию соответствия функций программы ее назначению;
* демонстрацию реализации требований к характеристикам программы;
* отображение надежности как индикатора качества программы.

А чего не может тестирование? Тестирование не может показать отсутствия дефектов (оно может показывать только присутствие дефектов). Важно помнить это (скорее печальное) утверждение при проведении тестирования.

Рассмотрим информационные потоки процесса тестирования. Они показаны на рис. 6.1.



**Рис. 6.1.** Информационные потоки процесса тестирования

На входе процесса тестирования три потока:

* текст программы;
* исходные данные для запуска программы;
* ожидаемые результаты.

Выполняются тесты, все полученные результаты оцениваются. Это значит, что реальные результаты тестов сравниваются с ожидаемыми результатами. Когда обнаруживается несовпадение, фиксируется ошибка — начинается отладка. Процесс отладки непредсказуем по времени. На поиск места дефекта и исправление может потребоваться час, день, месяц. Неопределенность в отладке приводит к большим трудностям в планировании действий.

После сбора и оценивания результатов тестирования начинается отображение качества и надежности ПО. Если регулярно встречаются серьезные ошибки, требующие проектных изменений, то качество и надежность ПО подозрительны, констатируется необходимость усиления тестирования. С другой стороны, если функции ПО реализованы правильно, а обнаруженные ошибки легко исправляются, может быть сделан один из двух выводов:

* качество и надежность ПО удовлетворительны;
* тесты не способны обнаруживать серьезные ошибки.

В конечном счете, если тесты не обнаруживают ошибок, появляется сомнение в том, что тестовые варианты достаточно продуманы и что в ПО нет скрытых ошибок. Такие ошибки будут, в конечном итоге, обнаруживаться пользователями и корректироваться разработчиком на этапе сопровождения (когда стоимость исправления возрастает в 60-100 раз по сравнению с этапом разработки).

Результаты, накопленные в ходе тестирования, могут оцениваться и более формальным способом. Для этого используют модели надежности ПО, выполняющие прогноз надежности по реальным данным об интенсивности ошибок.

Существуют 2 принципа тестирования программы:

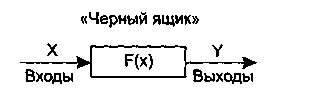
* функциональное тестирование (тестирование «черного ящика»);
* структурное тестирование (тестирование «белого ящика»).

### Тестирование «черного ящика»

**Известны:** функции программы.

**Исследуется:** работа каждой функции на всей области определения.

Как показано на рис. 6.2, основное место приложения тестов «черного ящика» — интерфейс ПО.



**Рис. 6.2.** Тестирование «черного ящика»

Эти тесты демонстрируют:

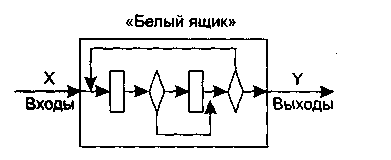
* как выполняются функции программ;
* как принимаются исходные данные;
* как вырабатываются результаты;
* как сохраняется целостность внешней информации.

При тестировании «черного ящика» рассматриваются системные характеристики программ, игнорируется их внутренняя логическая структура. Исчерпывающее тестирование, как правило, невозможно. Например, если в программе 10 входных величин и каждая принимает по 10 значений, то потребуется 1010 тестовых вариантов. Отметим также, что тестирование «черного ящика» не реагирует на многие особенности программных ошибок.

### Тестирование «белого ящика»

**Известна:** внутренняя структура программы.

**Исследуются:** внутренние элементы программы и связи между ними (рис. 6.3).



**Рис. 6.3.** Тестирование «белого ящика»

Объектом тестирования здесь является не внешнее, а внутреннее поведение программы. Проверяется корректность построения всех элементов программы и правильность их взаимодействия друг с другом. Обычно анализируются управляющие связи элементов, реже — информационные связи. Тестирование по принципу «белого ящика» характеризуется степенью, в какой тесты выполняют или покрывают логику (исходный текст) программы. Исчерпывающее тестирование также затруднительно. Особенности этого принципа тестирования рассмотрим отдельно.

Содержание лекции: виды контроля качества разрабатываемого ПО; ручной контроль; структурное, функциональное и оценочное тестирование; классификация ошибок; методы и средства отладки ПО.

Цель лекции: ознакомиться с видами и способами контроля и тестирования ПО, методами и средствами отладки программ.

Недостаточно выполнить проектирование и кодирование программного продукта, также необходимо обеспечить его соответствие требованиям и спецификациям. Многократно проводимые исследования показали, что чем раньше обнаруживаются те или иные несоответствия или ошибки, тем боль­ше вероятность их исправления и ниже его стои­мость [4]. Современные технологии разработки ПО преду­сматривают раннее обнаружение ошибок за счет выполнения контроля ре­зультатов всех этапов и стадий разработки. На начальных этапах кон­троль осуществляют вручную или с использованием CASE-средств, на последних - он принимает форму тестирования.

Тестирование- это процесс выполнения программы, целью которого является выявление ошибок. Никакое тестирование не может доказать отсут­ствие ошибок в сложном ПО, поскольку выполнение полного тестирования становится не­возможным и имеется вероятность, что остались невыявленные ошибки. Соблюдение основных правил тестирования и научно обоснованный подбор тестов может уменьшить их количество. Процесс разработки согласно современной модели жизненного цикла ПО предполагает три стадии тестирования: автономное тестирование компонентов ПО; комплексное тестирование разрабатываемого ПО; системное или оценочное тестирование на соответствие основным критериям качества. Для повышения качества тестирования рекомендуется соблюдать следу­ющие основные принципы:

а) предполагаемые результаты должны быть известны до тестирования;

б) следует избегать тестирования программы автором;

в) необходимо досконально изучать результаты каждого теста;

г) необходимо проверять действия программы на неверных данных;

д) необходимо проверять программу на неожиданные побочные эффекты на неверных данных.

Вероятность наличия необнаруженных ошибок в части программы пропорциональна количеству ошибок уже най­денных в этой части. Удачнымсчитают тест, ко­торый обнаруживает хотя бы одну ошибку. Формирование набора тестов имеет большое значение, поскольку тести­рование является одним из наиболее трудоемких этапов создания ПО. Доля стоимос­ти тестирования в общей стоимости разработки возрастает при увеличении сложности ПО и повышении требо­ваний к их качеству.

Существуют два принципиально различных подхода к формированию тестовых наборов: структурный и функциональный. Структурный подходбазируется на том, что известна структуратести¬руемого ПО, в том числе его алгоритмы («стеклян¬ный ящик»). Тесты строятся для проверки правильности реализации заданной логики в коде программы. Функциональный подход основывается на том, что структура ПО не известна («черный ящик»). В этом случае тесты строят, опираясь на функциональные спецификации. Этот подход называют также подходом, управляемым данными, так как при его использовании тесты строят на базе различных способов декомпозиции множества данных. Наборы тестов, полученные в соответствии с методами этих подходов, объединяют, обеспечивая всестороннее тестирование ПО.

Ручной контроль используют на ранних эта¬пах разработки. Все проектные решения анализируются с точки зрения их правильности и целесообразности как можно раньше, пока их можно легко пересмотреть. Различают статический и динамический подходы к ручному контролю. При статическомподходе анализируют структуру, управляющие и инфор¬мационные связи программы, ее входные и выходные данные. При динамическом - выполняют ручное тестирование (вручную моделируют про¬цесс выполнения программы на заданных исходных данных). Исходными данными для таких проверок являются: техническое зада¬ние, спецификации, структурная и функциональная схемы программного продукта, схемы отдельных компонентов, а для более поздних этапов - алгоритмы и тексты программ, а также тестовые наборы. Доказано, что ручной контроль способствует существенному увеличе¬нию производительности и повышению надежности программ и с его помо¬щью можно находить от 30 до 70 % ошибок логического проектирования и кодирования. Основными методами ручного контроля являются: инспекции исходного текста, сквозные просмотры, проверка за столом, оценки программ.

В основе структурного тестированиялежит концепция максимально полного тестирования всех маршрутов, предусмотренных алгоритмом (последовательности операторов программы, выполняемых при конкретном варианте исходных данных). Недостатки: построенные тесто¬вые наборы не обнаруживают пропущенных маршрутов и ошибок, зависящих от заложенных данных; не дают гарантии, что программа правильна.

Другим способом проверки программ является функциональное тестирование: про-грамма рассматривается как «черный ящик», целью тестирования является выяснение обстоятельств, когда поведение программы не соответствует спецификации. Для обнаружения всех ошибок необходимо выполнить исчерпывающее тестирование (при всех возможных наборах данных), что для большинства случаев невоз¬можно. Поэтому обычно выполняют «разумное» или «приемлемое» тестиро¬вание, ограничивающееся прогонами программы на небольшом под¬множестве всех возможных входных данных. При функциональном тестировании различают следующие методы фор¬мирования тестовых наборов: эквивалентное разбиение; анализ граничных значений; анализ причинно-следственных связей; предположение об ошибке.

При комплексном тестирова¬нии используют тесты, построенные по методам эквивалентных классов, граничных условий и предположении об ошибках, поскольку структурное тестирование для него не при¬менимо. Одним из самых сложных является вопрос о завершении тестирования, так как невозможно гарантировать, что в программе не осталось ошибок. Часто тестирование завершают потому, что закончилось время, отведен¬ное на его выполнение. Его сворачивают, обходясь минимальным тестированием [15], которое предпо¬лагает: тестирование граничных значений, тщательную проверку руководства, тестирование минимальных конфигураций технических средств, возможности редактирования команд и повторения их в любой последовательности, устойчивости к ошибкам пользователя.

После завершения комплексного тестирования приступают к оценочному тестированию, целью которого является поиск несоответствий техническому заданию. Оценочное тестирование включает тестирование: удобства использования, на предельных объемах, на предельных нагрузках, удобства эксплуатации, защиты, производительности, требований к памяти, конфигурации оборудования, совместимости, удобства установки, удобства обслуживания, надежности, восстановления, документации, процедуры.

Отладка- это процесс локализации (определения оператора программы, выполнение которого вызвало нарушение вычислительного процесса) и исправления ошибок, обнаружен¬ных при тестировании ПО. Для исправления ошиб¬ки необходимо определить ее причину. Отладка требует от программиста глубоких знаний специфики управления используемыми техническими средствами, операционной системы, среды и языка программирования, реализуемых процессов, природы и специфики ошибок, методик отладки и соответствующих программных средств; психологически дискомфортна (нужно искать собственные ошибки в условиях ограниченного времени); оставляет возможность взаимовлияния ошибок в разных частях программы. Четко сформулированные методики отладки отсутствуют. Различают:

а) синтаксические ошибки – сопровождаются комментарием с указанием их мес-тоположения, фиксируются компилятором (транслятором) при выполнении синтаксического и частично се¬мантического анализа;

б) ошибки компоновки - обнаруживаются компоновщиком (редакто¬ром связей) при объединении модулей программы;

в) ошибки выполнения - обнаруживаются аппаратными средствами, операционной системой или пользователем при выполнении программы, проявляются разными способами и в свою очередь делятся на группы:

1) ошибки определения исходных данных (ошибки передачи, ошибки преобразования, ошибки перезаписи и ошиб¬ки данных);

2) логические ошибки проектирования (неприменимый метод, неверный алгоритм, неверная структура данных, другие) и кодирования (ошибки некорректного использования переменных, вычислений, межмодульного интерфейса, реализации алгоритма, другие);

3) ошибки накопления погрешностей результатов вычислений (игнорирование ограничений разрядной сетки и способов уменьшения погрешности).

Отладка программы в любом случае предполагает обдумывание и логи¬ческое осмысление всей имеющейся информации об ошибке. Большинство ошибок можно обнаружить по косвенным признакам посредством тщатель¬ного анализа текстов программ и результатов тестирования без получения дополнительной информации с помощью следующих методов:

а) ручного тестирования (при обнаружении ошибки нужно выполнить те¬стируемую программу вручную, используя тестовый набор, при работе с ко¬торым была обнаружена ошибка);

б) индукции (основан на тща¬тельном анализе симптомов ошибки, которые могут проявляться как неверные результаты вычислений или как сообщение об ошибке);

в) дедукции (вначале формируют множество причин, которые могли бы вызвать данное проявление ошибки, а затем анали¬зируя причины, исключают те, которые противоречат имеющимся данным);

г) обратного прослеживания (для точки вы¬вода неверного результата строится гипотеза о значени¬ях основных переменных, которые могли бы привести к получению данного результата, а затем, исходя из этой гипотезы, делают предположения о значениях переменных в предыдущей точке).

Для получения дополнительной информации об ошибке выпол¬няют добавочные тесты и используют специальные методы и средства: отладочный вывод; интегрированные средства отладки; независимые отладчики.

Общая методика отладки программных продуктов, написанных для выполнения в операционных системах MS DOS и Win32:

1 этап - изучение проявления ошибки;

2 этап – определение локализации ошибки;

3 этап - определение причины ошибки;

4 этап — исправление ошибки;

5 этап - повторное тестирование.

Процесс отладки можно существенно упрос¬тить, если следовать основным рекомендациям структурного подхода к про¬граммированию:

а) программу наращивать «сверху-вниз», от интерфейса к обрабатываю¬щим подпрограммам, тестируя ее по ходу добавления подпрограмм;

б) выводить пользователю вводимые им данные для контроля и прове¬рять их на допустимость сразу после ввода;

в) предусматривать вывод основных данных во всех узловых точках ал¬горитма (ветвлениях, вызовах подпрограмм).

*Тестирование базового пути* — это способ, который основан на принципе «белого ящика». Автор этого способа — Том МакКейб (1976) [49].

Способ тестирования базового пути дает возможность:

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

Тестовые варианты разрабатываются для проверки базового множества путей (маршрутов) в программе. Они гарантируют однократное выполнение каждого оператора программы при тестировании.

### Потоковый граф

Для представления программы используется потоковый граф. Перечислим его особенности.

1. Граф строится отображением управляющей структуры программы. В ходе отображения закрывающие скобки условных операторов и операторов циклов (end if; end loop) рассматриваются как отдельные (фиктивные) операторы.

2. Узлы (вершины) потокового графа соответствуют линейным участкам программы, включают один или несколько операторов программы.

3. Дуги потокового графа отображают поток управления в программе (передачи управления между операторами). Дуга — это ориентированное ребро.

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

1. Предикатные узлы соответствуют простым условиям в программе. Составное условие программы отображается в несколько предикатных узлов. Составным называют условие, в котором используется одна или несколько булевых операций (OR, AND).
2. Например, фрагмент программы

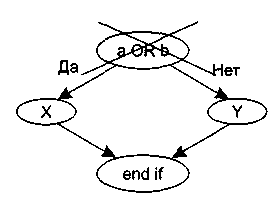
if a OR b

then x

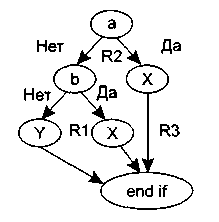
else у

end if;

вместо прямого отображения в потоковый граф вида, показанного на рис. 6.4, отображается в преобразованный потоковый граф (рис. 6.5).



**Рис. 6.4.** Прямое отображение в потоковый граф



**Рис. 6.5.** Преобразованный потоковый граф

6. Замкнутые области, образованные дугами и узлами, называют регионами.

7. Окружающая граф среда рассматривается как дополнительный регион. Например, показанный здесь граф имеет три региона — Rl, R2, R3.

**Пример 1.** Рассмотрим процедуру сжатия:

процедура сжатие

1 выполнять пока нет EOF

1 читать запись;

2 если запись пуста

3 то удалить запись:

4 иначе если поле а >= поля b

5 то удалить b;

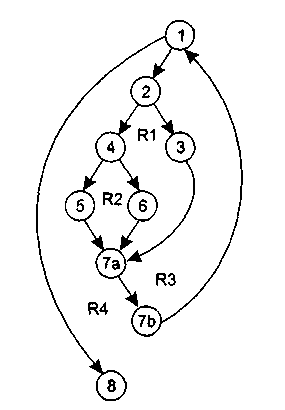
6иначе удалить а;

7а конец если;

7а конец если;

7b конец выполнять;

8 конец сжатие;



**Рис. 6.6.** Преобразованный потоковый граф процедуры сжатия

Она отображается в потоковый граф, представленный на рис. 6.6. Видим, что этот потоковый граф имеет четыре региона.

### Цикломатическая сложность

Цикломатическая сложность — метрика 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, которое гарантирует однократное выполнение всех операторов.

Независимым называется любой путь, который вводит новый оператор обработки или новое условие. В терминах потокового графа независимый путь должен содержать дугу, не входящую в ранее определенные пути.

**ПРИМЕЧАНИЕ**

Путь начинается в начальном узле, а заканчивается в конечном узле графа. Независимые пути формируются в порядке от самого короткого к самому длинному.

Перечислим независимые пути для потокового графа из примера 1:

**Путь 1:** 1-8.

**Путь 2:** 1-2-3-7а-7b-1-8.

**Путь 3:** 1-2-4-5-7а-7b-1-8.

Путь 4: 1-2-4-6-7а-7b-1-8.

Заметим, что каждый новый путь включает новую дугу.

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 по False-ветви;

2) мощность базового множества равна цикломатической сложности потокового графа.

Значение 2-го свойства трудно переоценить — оно дает априорную оценку количества независимых путей, которое имеет смысл искать в графе.

Цикломатическая сложность вычисляется одним из трех способов:

1) цикломатическая сложность равна количеству регионов потокового графа;

2) цикломатическая сложность определяется по формуле

*V(G)-E-N+2,*

где *Е —* количество дуг, *N —* количество узлов потокового графа;

3) цикломатическая сложность формируется по выражению *V(G) =p+* 1, где *р* — количество предикатных узлов в потоковом графе *G.*

Вычислим цикломатическую сложность графа из примера 1 каждым из трех способов:

1) потоковый граф имеет 4 региона;

2) *V(G)* = 11 дуг - 9 узлов + 2 = 4;

3) *V(G)* = 3 предикатных узла +1=4.

Таким образом, цикломатическая сложность потокового графа из примера 1 равна четырем.

### Шаги способа тестирования базового пути

Для иллюстрации шагов данного способа используем конкретную программу — процедуру вычисления среднего значения:

процедура сред;

1 i := 1;

1 введено := 0;

1 колич := 0;

1 сум := 0;

вып пока 2 -вел( i ) <> stop и введено <=500 - 3

4 введено:= введено + 1;

если 5 -вел( i ) >= мин и вел( i ) <= макс - 6

7 то колич := колич + 1;

7 сум := сум + вел( i );

8 конец если;

8 i := i + 1;

9 конец вып;

10 если колич > 0

11 то сред := сум / колич;

12 иначе сред := stop;

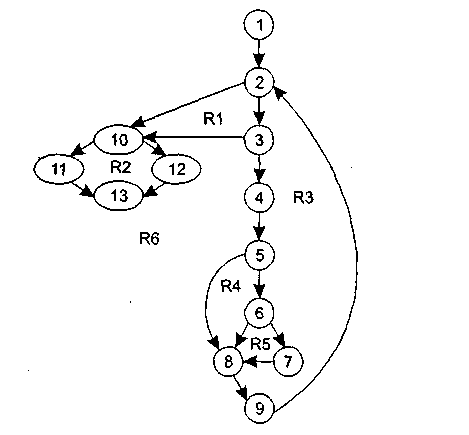
13 конец если;

13 конец сред;

Заметим, что процедура содержит составные условия (в заголовке цикла и условном операторе). Элементы составных условий для наглядности помещены в рамки.

*Шаг 1.* На основе текста программы формируется потоковый граф:

* нумеруются операторы текста (номера операторов показаны в тексте процедуры);
* производится отображение пронумерованного текста программы в узлы и вершины потокового графа (рис. 6.7).



**Рис. 6.7.** Потоковый граф процедуры вычисления среднего значения

*Шаг 2.* Определяется цикломатическая сложность потокового графа — по каждой из трех формул:

1) *V(G) =* 6 регионов;

2) *V(G)* = 17 дуг - 13 узлов + 2 = 6;

3) *V(G)* = 5 предикатных узлов + 1 = 6.

*Шаг 3.* Определяется базовое множество независимых линейных путей:

**Путь 1:** 1-2-10-11-13; /вел=stор, колич>0.

**Путь 2:** 1-2-10-12-13;/вел=stop, колич=0.

**Путь 3:** 1-2-3-10-11-13; /попытка обработки 501-й величины.

**Путь 4:** 1-2-3-4-5-8-9-2-... /вел<мин.

**Путь 5:** 1-2-3-4-5-6-8-9-2-... /вел>макс.

**Путь 6:** 1-2-3-4-5-6-7-8-9-2-... /режим нормальной обработки.

**ПРИМЕЧАНИЕ**

Для удобства дальнейшего анализа по каждому пути указаны условия запуска. Точки в конце путей 4, 5, 6 указывают, что допускается любое продолжение через остаток управляющей структуры графа.

*Шаг 4.* Подготавливаются тестовые варианты, инициирующие выполнение каждого пути.

Каждый тестовый вариант формируется в следующем виде:

*Исходные данные (ИД):*

*Ожидаемые результаты (ОЖ.РЕЗ.):*

Исходные данные должны выбираться так, чтобы предикатные вершины обеспечивали нужные переключения — запуск только тех операторов, которые перечислены в конкретном пути, причем в требуемом порядке.

Определим тестовые варианты, удовлетворяющие выявленному множеству независимых путей.

Тестовый вариант для пути 1 **ТВ1**:

*ИД:* вел(*k*) = допустимое значение, где *k < i*;вел(*i*) = stop, где 2 < i < 500.

*ОЖ.РЕЗ.:* корректное усреднение основывается на *k* величинах и правильном подсчете.

**ПРИМЕЧАНИЕ**

Путь не может тестироваться самостоятельно, а должен тестироваться как часть путей 4, 5, 6 (трудности проверки 11-го оператора).

Тестовый вариант для пути 2 **ТВ2**:

*ИД*: вел(1)=stор.

*ОЖ.РЕЗ.:* сред=stор, другие величины имеют начальные значения.

Тестовый вариант для пути 3 **ТВЗ**:

*ИД:* попытка обработки 501-й величины, первые 500 величин должны быть правильными.

*ОЖ.РЕЗ.:* корректное усреднение основывается на *k* величинах и правильном подсчете.

Тестовый вариант для пути 4 **ТВ4**:

*ИД:* вел(*i*)=допустимое значение, где *i* ≤ 500; вел(*k*) < мин, где *k < i.*

*ОЖ.РЕЗ.:* корректное усреднение основывается на *k* величинах и правильном подсчете.

Тестовый вариант для пути 5 **ТВ5**:

*ИД:* вел(*i*)=допустимое значение, где *i* ≤ 500; вел(*k*) > макс, где *k* < *i*.

*ОЖ.РЕЗ.:* корректное усреднение основывается на *п* величинах и правильном подсчете.

Тестовый вариант для пути 6 **ТВ6**:

*ИД:* вел(*i*)=допустимое значение, где *i* ≤ 500.

*ОЖ.РЕЗ.:* корректное усреднение основывается на *п* величинах и правильном подсчете.

Реальные результаты каждого тестового варианта сравниваются с ожидаемыми результатами. После выполнения всех тестовых вариантов гарантируется, что все операторы программы выполнены по меньшей мере один раз.

Важно отметить, что некоторые независимые пути не могут проверяться изолированно. Такие пути должны проверяться при тестировании другого пути (как часть другого тестового варианта).