2.1. Введение.

Понятия процесса программирования качественно изменились.

Производство программ приобрело массовый характер, существенно

увеличились их объем и сложность. Разработка программных комп-

лексов потребовала значительных усилий больших коллективов

специалистов. Программы перестали быть только вычислительными

и начали выполнять важнейшие функции по управлению и обработке

информации в различных отраслях.

Развитие и применение технологий проектирования комплексов

программ приводит к необходимости измерения и сравнения их эф-

фективности прежде всего по степени влияния на качество прог-

раммного продукта.

Обеспечение высокого качества сложных комплексов программ

связано со значительными затратами труда разработчиков. Затра-

ты на создание программ быстро увеличиваются при возрастании

требований, причем для сложных комплексов весьма сложно дос-

тичь высокого качества функционирования, и после обеспечения

общей работоспособности могут понадобится годы труда для полу-

чения необходимых показателей качества. Поэтому уже сегодня

требуются методы и средства, которые позволили бы заметно по-

высить качество программ программ при относительно невысоких

затратах труда.

2.2. Обоснование выбора технологии тестирования.

Как известно, при создании типичного программного проекта

около 50% общего времени и более 50% общей стоимости расходу-

ется на проверку (тестирование) разрабатываемой программы или

системы. Кроме того, доля стоимости тестирования в общей стои-

мости программ имеет тенденцию возрастать при увеличении слож-

ности комплексов программ и повышения требований к их качест-

ву.

Учитывая это, при отработке технологии тестирования прог-

рамм следует четко выделять определенное (по возможности не

очень большое) число правил отладки, обеспечивающих высокое

качество программного продукта и снижающих затраты на его соз-

дание.

Тестирование - это процесс исполнения программы с целью

обнаружения ошибок. Одним из способов изучения поставленного

вопроса является исследование стратегии тестирования, называе-

мой стратегией черного ящика, тестированием с управлением по

данным, или тестированием с управлением по входу-выходу. При

использовании этой стратегии программа рассматривается как

черный ящик. Тестовые данные используются только в соответст-

вии со спецификацией программы (т.е. без учета знаний о ее

внутренней структуре).

При таком подходе обнаружение всех ошибок в программе яв-

ляется критерием исчерпывающего входного тестирования. Послед-

нее может быть достигнуто, если в качестве тестовых наборов

использовать все возможные наборы входных данных. Следователь-

но, мы приходим к выводу, что для исчерпывающего тестирования

программы требуется бесконечное число тестов, а значит постро-

ение исчерпывающего входного теста невозможно. Это подтвержда-

ется двумя аргументами: во-первых, нельзя создать тест, гаран-

тирующий отсутствие ошибок; во-вторых, разработка таких тес-

тов противоречит экономическим требованиям. Поскольку исчерпы-

вающее тестирование исключается, нашей целью должна стать мак-

симизация результативности вложения капиталовложений в тести-

рование (максимизация числа ошибок, обнаруживаемых одним тес-

том). Для этого необходимо рассматривать внутреннюю структуру

программы и делать некоторые разумные, но, конечно, не облада-

ющие полной гарантией достоверности предположения.

Стратегия белого ящика, или стратегия тестирования, управ-

ляемого логикой программы, позволяет исследовать внутреннюю

структуру программы. В этом случае тестирующий получает тесто-

вые данные путем анализа логики программы.

Сравним способ построения тестов при данной стратегии с

исчерпывающим входным тестированием стратегии черного ящика.

Неверно предположение, что достаточно построить такой набор

тестов, в котором каждый оператор исполняется хотя бы один

раз. Исчерпывающему входному тестированию может быть поставле-

но в соответствие исчерпывающее тестирование маршрутов. Подра-

зумевается, что программа проверена полностью, если с помощью

тестов удается осуществить выполнение этой программы по всем

возможным маршрутам ее потока (графа) передач управления.

Последнее утверждение имеет два слабых пункта: во-первых,

число не повторяющих друг друга маршрутов - астрономическое;

во-вторых, даже если каждый маршрут может быть проверен, сама

программа может содержать ошибки (например, некоторые маршруты

пропущены).

В результате всех изложенных выше замечаний можно отме-

тить, что ни исчерпывающее входное тестирование ни исчерпываю-

щее тестирование маршрутов не могут стать полезными стратегия-

ми, потому что оба они не реализуемы. Поэтому реальным путем,

который позволит создать хорошую, но, конечно не абсолютную

стратегию, является сочетание тестирования программы несколь-

кими методами.

2.3. Разработка технологического процесса тестирования.

Если отказаться от тестирования всех путей, то можно пока-

зать, что критерием покрытия является выполнение каждого опе-

ратора программы по крайней мере один раз.

В качестве примера тестирования возьмем модуль Param.

Предназначение модуля - разбирать командную строку с парамет-

рами на отдельные параметры.

Объектом тестирования изберем правило ParamStr объекта

Parameters.

function Parameters.ParamStr(ParamNum : byte) : string;

begin

if ParamNum = 0 then

if Delux then

ParamStr:=''

else

if Lo(DosVersion) >= 3 then

ParamStr:=system.ParamStr(0)

else

ParamStr:=''

else

ParamStr:=OptionStr(ParamNum);

end;

Схема алгоритма этой функции:

--------------------¬

¦ Начало ¦

L---------T----------

¦

/ \

/ \ нет

/ParamNum \ ----------------¬

\ = 0 / ¦

\ / ----------+---------¬

\ /да ¦ ParamStr = ¦

¦ ¦OptionStr(ParamNum)¦

/ \ L---------T----------

да / \ ¦

-<-----------/ Delux \ ¦

¦ \ = true / ¦

----------+-------¬ \ / ¦

¦ ParamStr = '' ¦ \ /нет ¦

L---------T-------- ¦ ¦

¦ / \ ¦

¦ / Lo( \ нет ¦

¦ /DosVersion --------------¬ L--------¬

¦ \ ) >= 3 / ¦ ¦

¦ \ / ¦ ¦

¦ \ /да ¦ ¦

¦ -----------+---------¬ ---------+--------¬ ¦

¦ ¦ ParamStr = System.¦ ¦ ParamStr = '' ¦ ¦

¦ ¦ ParamStr(0) ¦ L--------T--------- ¦

¦ L----------T---------- ¦ ¦

L---------------->+<-------------------+<----------

----------+---------¬

¦ Конец ¦

L--------------------

Рис 2.1.

Табл. 2.1.

г===T==================T===================T====================¬

¦ N ¦ Входные данные ¦Ожидаемый результат¦Полученный результат¦

¦---+------------------+-------------------+--------------------¦

¦ 1 ¦ ParamNum = 1 ¦ ParamStr = ¦ ParamStr = ¦

¦ ¦ ¦OptionStr(ParamNum)¦OptionStr(ParamNum) ¦

¦---+------------------+-------------------+--------------------¦

¦ 2 ¦ ParamNum = 0 ¦ ParamStr = '' ¦ ParamStr = '' ¦

¦ ¦ Delux = true ¦ ¦ ¦

¦---+------------------+-------------------+--------------------¦

¦ 3 ¦ ParamNum = 0 ¦ParamStr = ¦ParamStr = ¦

¦ ¦ Delux = false ¦ System.ParamStr(0)¦ System.ParamStr(0) ¦

¦ ¦ Lo(DosVersion)=3 ¦ ¦ ¦

¦---+------------------+-------------------+--------------------¦

¦ 4 ¦ ParamNum = 0 ¦ParamStr = '' ¦ParamStr = '' ¦

¦ ¦ Delux = false ¦ ¦ ¦

¦ ¦ Lo(DosVersion)=2 ¦ ¦ ¦

L===¦==================¦===================¦====================-

Данный критерий тестирования хуже, чем кажется на первый

взгляд. Например, если условие Lo(DosVersion) >= 3 будет оши-

бочно записано Lo(DosVersion) > 3. При тестировании по данно-

му критерию эта ошибка не будет обнаружена.

Более сильный критерий покрытия логики программы известен

как покрытие решений, или покрытие переходов. Согласно данно-

му критерию должно быть записано достаточное число тестов, та-

кое, что каждое решение на этих тестах примет значение истина

и ложь по крайней мере один раз.

Можно показать, что покрытие решений обычно удовлетворяет

критерию покрытия операторов. Поскольку каждый оператор лежит

на некотором пути, исходящем из оператора перехода, либо из

точки входа программы, при выполнении каждого направления пе-

рехода каждый оператор должен быть выполнен. Следовательно,

тесты приведенные выше подходят и для этого критерия.

Однако существуют исключения, например, оператор case. В

этом операторе возможны не двузначные решения.

CASE условие OF

m1 : оператор1;

m2 : оператор2;

m3 : оператор3

ELSE

m4 : оператор4

END

Критерием для таких случаев является выполнение каждого

возможного результата всех решений по крайней мере один раз.

Лучшим критерием по сравнению с предыдущим является покры-

тие условий. В этом случае записывают число тестов, достаточ-

ное для того, чтобы все возможные результаты каждого условия

в решении выполнялись по крайней мере один раз.

Рассмотрим пример на функции OptionStr.

function Parameters.OptionStr(ParamNum : byte) : string;

var

I, Len : Byte;

begin

Len := 0;

I := OptPosition(ParamNum);

if I <> 0 then

while (I <= SLen) and not (ParStr[I] in OptDelim) do

begin

Inc(Len);

OptionStr[Len] := ParStr[I];

Inc(I);

end;

OptionStr[0] := Char(Len);

end;

Алгоритм этой функции:

--------------------¬

¦ Начало ¦

L---------T----------

----------+---------¬

¦ Len = 0; ¦

¦ I = OptPosition( ¦

¦ ParamNum) ¦

L---------T----------

/ \

/ \ да

/ I = 0 \ ----------------¬

\ / ¦

\ / ¦

\ /нет ¦

----------------+ ¦

¦ / \ ¦

¦ / \ ¦

¦ /I <= SLen\ да ¦

¦ / и не \ ------------->+

¦ \ParStr(I) в / ¦

¦ \OptDelim / ¦

¦ \ / ¦

¦ \ / нет ¦

¦ ----------+---------¬ ¦

¦ ¦ Len = Len + 1; ¦ ¦

¦ ¦ OptionStr(Len) = ¦ ¦

¦ ¦ ParStr(I) ¦ ¦

¦ L---------T---------- ¦

L---------------- ¦

------------------------

----------+---------¬

¦ Конец ¦

L--------------------

Рис 2.2.

Функция содержит три условия:

I=0, I<=SLen, not (ParStr[i] in OptDelim).

Следовательно, требуется достаточное число тестов, такое,

чтобы реализовать ситуации, где I=0, I<>0 в первом условии и

I<=SLen, I>SLen, (ParStr[i] in OptDelim)=true, (ParStr[i] in

OptDelim)=false во втором условии.

Тесты, удовлетворяющие критерию покрытия условий пиведены

в таблице 2.2. (пусть стока параметров имеет вид: MAIN.GRM

/Q/P, SLen=13, ParamNum=1):

Табл. 2.2.

г===T==================T===================T====================¬

¦ N ¦ Входные данные ¦Ожидаемый результат¦Полученный результат¦

¦---+------------------+-------------------+--------------------¦

¦ 1 ¦ I = 0 ¦ OptionStr(0) = 0 ¦ OptionStr(0) = 0 ¦

¦ ¦ ¦ ¦ ¦

¦---+------------------+-------------------+--------------------¦

¦ 2 ¦ I = 1 ¦ OptionStr(0) = 0 ¦ OptionStr(0) = 0 ¦

¦ ¦ (ParStr[i] in ¦ ¦ ¦

¦ ¦ OptDelim) = true¦ ¦ ¦

¦---+------------------+-------------------+--------------------¦

¦ 3 ¦ I = 1 ¦ OptionStr(0) = 8 ¦ OptionStr(0) = 8 ¦

¦ ¦ (ParStr[i] in ¦ ¦ ¦

¦ ¦ OptDelim)=false ¦ ¦ ¦

¦---+------------------+-------------------+--------------------¦

¦ 4 ¦ I = 11 ¦ OptionStr(0) = 0 ¦ OptionStr(0) = 0 ¦

¦ ¦ (ParStr[i] in ¦ ¦ ¦

¦ ¦ OptDelim) = true¦ ¦ ¦

¦---+------------------+-------------------+--------------------¦

¦ 5 ¦ I = 11 ¦ OptionStr(0) = 0 ¦ OptionStr(0) = 0 ¦

¦ ¦ (ParStr[i] in ¦ ¦ ¦

¦ ¦ OptDelim)=false ¦ ¦ ¦

L===¦==================¦===================¦====================-

Хотя применение критерия покрытия условий на первый

взгляд удовлетворяет критерию покрытия решений, это не всегда

так. Если тестируется решение

if A and B then ...

то при критерии покрытия условий требовались бы два теста:

A = true, B = false и A = false, B = true. Но в этом случае

не выполнялось бы then-предложение оператора if.

Существует еще один критерий, названный покрытием реше-

ний/условий. Он требует такого достаточного набора тестов,

чтобы все возможные результаты каждого условия в решении вы-

полнялись по крайней мере один раз, все результаты каждого ре-

шения выполнялись по крайней мере один раз и каждой точке вхо-

да передавалось управление по крайней мере один раз.

Недостатком критерия покрытия решений/условий является не-

возможность его применения для выполнения всех результатов

всех условий; часто подобное выполнение имеет место в следст-

вии того, что определенные условия скрыты другими условиями.

Например, если условие AND есть ложь, то никакое из последую-

щих условий в выражении не будет выполнено. Аналогично, если

условие OR есть истина, то никакое из последующих условий не

будет выполнено. Следовательно, критерии покрытия условий и

покрытия решений/условий недостаточно чувствительны к ошибкам

в логических выражениях.

Критерием, который решает эти и некоторые другие пробле-

мы, является комбинаторное покрытие условий. Он требует созда-

ния такого числа тестов, чтобы все возможные комбинации резу-

льтатов условия в каждом решении и все точки входа выполня-

лись по крайней мере один раз.

Рассмотрим правило CheckTreeNil в модуле TmObejct объекта

Main.

procedure Main.CheckTreeNil;

var

tn : boolean;

begin

tn := (GetPtrOfClass(SCl)=nil) and

(GetPtrOfClass(UCl)=nil) and

(GetPtrOfClass(ACl)=nil);

if tn then Error('не найден ни один нетерминал');

end;

Алгоритм процедуры:

--------------------¬

¦ Начало ¦

L---------T----------

¦

/\

/ \

/ \

/ G(SCl)=nil \

/ и \ нет

/ G(UCl)=nil \-----------¬

\ и / ¦

\ G(ACl)=nil / ¦

\ / ¦

\ / ¦

\ / да ¦

¦ ¦

--------------------+------------------¬ ¦

¦ Error('не найден ни один нетерминал')¦ ¦

L-------------------T------------------- ¦

+<----------------------

----------+---------¬

¦ Конец ¦

L--------------------

Рис 2.3.

Для того, чтобы протестировать эту процедуру необходимо

восемь тестов, хотя она покрывается всего двумя путями.

Табл. 2.3.

г===T===========================T============T============¬

¦ N ¦ Входные данные ¦ Ожидаемый ¦ Полученный ¦

¦ ¦ ¦ результат ¦ результат ¦

¦---+---------------------------+------------+------------¦

¦ ¦ GetPtrOfClass(SCl) = nil ¦ ¦ ¦

¦ 1 ¦ GetPtrOfClass(UCl) = nil ¦ tn = true ¦ tn = true ¦

¦ ¦ GetPtrOfClass(ACl) = nil ¦ ¦ ¦

¦---+---------------------------+------------+------------¦

¦ ¦ GetPtrOfClass(SCl) <> nil ¦ ¦ ¦

¦ 2 ¦ GetPtrOfClass(UCl) = nil ¦ tn = false ¦ tn = false ¦

¦ ¦ GetPtrOfClass(ACl) = nil ¦ ¦ ¦

¦---+---------------------------+------------+------------¦

¦ ¦ GetPtrOfClass(SCl) = nil ¦ ¦ ¦

¦ 3 ¦ GetPtrOfClass(UCl) <> nil ¦ tn = false ¦ tn = false ¦

¦ ¦ GetPtrOfClass(ACl) = nil ¦ ¦ ¦

¦---+---------------------------+------------+------------¦

¦ ¦ GetPtrOfClass(SCl) <> nil ¦ ¦ ¦

¦ 4 ¦ GetPtrOfClass(UCl) <> nil ¦ tn = false ¦ tn = false ¦

¦ ¦ GetPtrOfClass(ACl) = nil ¦ ¦ ¦

¦---+---------------------------+------------+------------¦

¦ ¦ GetPtrOfClass(SCl) = nil ¦ ¦ ¦

¦ 5 ¦ GetPtrOfClass(UCl) = nil ¦ tn = false ¦ tn = false ¦

¦ ¦ GetPtrOfClass(ACl) <> nil ¦ ¦ ¦

¦---+---------------------------+------------+------------¦

¦ ¦ GetPtrOfClass(SCl) <> nil ¦ ¦ ¦

¦ 6 ¦ GetPtrOfClass(UCl) = nil ¦ tn = false ¦ tn = false ¦

¦ ¦ GetPtrOfClass(ACl) <> nil ¦ ¦ ¦

¦---+---------------------------+------------+------------¦

¦ ¦ GetPtrOfClass(SCl) = nil ¦ ¦ ¦

¦ 7 ¦ GetPtrOfClass(UCl) <> nil ¦ tn = false ¦ tn = false ¦

¦ ¦ GetPtrOfClass(ACl) <> nil ¦ ¦ ¦

¦---+---------------------------+------------+------------¦

¦ ¦ GetPtrOfClass(SCl) <> nil ¦ ¦ ¦

¦ 8 ¦ GetPtrOfClass(UCl) <> nil ¦ tn = false ¦ tn = false ¦

¦ ¦ GetPtrOfClass(ACl) <> nil ¦ ¦ ¦

L===¦===========================¦============¦============-

В случае циклов число тестов для удовлетворения критерию

комбинаторного покрытия условий обычно больше, чем число пу-

тей.

Легко видеть, что набор тестов, удовлетворяющий критерию

комбинаторного покрытия условий, удовлетворяет также и крите-

риям покрытия решений, покрытия условий и покрытия решений/ус-

ловий.

Таким образом, для программ, содержащих только одно усло-

вие на каждое решение, минимальным является критерий, набор

тестов которого:

- вызывает выполнение всех результатов каждого решения по

крайней мере один раз;

- передает управление каждой точке входа (например, опера-

тор CASE).

Для программ, содержащих решения, каждое из которых имеет

более одного условия, минимальный критерий состоит из набора

тестов, вызывающих всех возможных комбинаций результатов усло-

вий в каждом решении и передающих управление каждой точке вхо-

да программы по крайней мере один раз.

В свете всего вышеизложенного, можно изобразить алгоритм

выбора минимального критерия, по которому необходимо тестиро-

вать программу (см. рис. 2.4.).

--------------------¬

¦ Начало ¦

L---------T----------

--------------------------------->+

¦ ----------+---------¬

¦ ¦ Выбрать оператор ¦

¦ ¦ условного перехода¦

¦ L---------T----------

¦ /\

¦ / \

¦ / \ нет

¦ /Это оператор\ ---------¬

¦ \ IF / ¦

¦ \ / ¦

¦ \ / ¦

¦ \/да ¦

¦ ¦ ¦

¦ /\ ¦

¦ / \ ¦

¦ да /Условие \ нет ¦

¦ ----------/ содержит \--------->+

¦ ¦ \более одного/ ¦

¦ ¦ \ комп-та/ ¦

¦ ¦ \ / ¦

¦ ¦ \/ ¦

¦ ----------------+---------------¬ ----------------+-------------¬

¦ ¦ Набор тестов, вызывающий все ¦ ¦ Набор тестов, вызывающий ¦

¦ ¦ возможные комбинации резуль-в ¦ ¦ выполнение всех результатов ¦

¦ ¦ условий в каждом решении не ¦ ¦ каждого решения не менее ¦

¦ ¦ менее одного раза. ¦ ¦ одного раза. ¦

¦ L---------------T---------------- L---------------T--------------

¦ L--------------->T<----------------

¦ /\

¦ /Это \

¦ нет /последн.\

L---------------------------/ оператор \

\ условного /

\перехода/

\ /

\/да

----------+---------¬

¦ Конец ¦

L--------------------

Рис 2.4.