Лекция 6. Стратегии проектирования тестов

Для того, чтобы обеспечить систематическое обнаружение различных классов ошибок (охватить, покрыть тестами максимальное количество аспектов создаваемой системы) и при этом минимизировать затраты на тестирование, разработаны две ***стратегии проектирования*** тестовых вариантов: стратегия «черного ящика» и стратегия «белого (прозрачного) ящика»

6.1. Стратегия "черного ящика"

Цель стратегии **черного ящика** – выявить ситуации, в которых поведение программы не соответствует спецификации. При этом программное изделие рассматривается как «черный ящик», внутреннее устройство которого не известно и чье поведение можно описать множеством входных воздействий и соответствующих им реакций.

В числе основных **критериев «черного ящика»** следующие:

1. **покрытие функций**: для каждой из функций, реализуемых программой, требуется подготовить и выполнить хотя бы один тест;
2. **покрытие классов входных данных**. Критерий требует выполнения хотя бы одного теста для каждого класса входных данных. Деление входных данных на классы осуществляется таким образом, чтобы все представители одного класса были равнозначны с точки зрения проверки правильности программы;
3. **покрытие классов выходных данных**. Проверяются выходные данные. Необходимо такое количество и тип тестовых данных, чтобы каждый допустимый результат был получен хотя бы один раз;
4. **покрытие области допустимых значений.** Речь о значениях переменных. Программа должна реагировать адекватно и на допустимые, и на недопустимые значения. Покрытие области допустимых значений означает, что выполнено хотя бы по одному тесту с нормальными, экстремальными (на границах класса) и исключительными (за пределами класса) значениями данной переменной. Аналогично, если переменная представляет собой набор данных (массив, строка символов и т.п.) тестами покрывают допустимое количество элементов в наборе: а) нормальная длина, б) ненормальная длина (пустой, слишком короткий, слишком длинный набор) и в) граничные значения (набор минимально возможной, максимально возможной длины).

Для задач сортировки и поиска экстремумов нужно тестирование **упорядоченности набора данных**. В этом случае необходимы тесты для проверки упорядоченности данных, направления упорядоченности (по возрастанию или убыванию), наличия повторяющихся значений и экстремумов.

Следует обратить внимание, что перечисленные критерии связаны между собой. Тест, покрывающий одну из функций, частично соответствует и остальным критериям, т.к. обычно каждой из функций ПП соответствует свои классы входных и выходных данных, которые выражаются через переменные, принимающие какие-то значения.

В числе **методов** формирования тестовых наборов по критериям «черного ящика» эквивалентное разделение, анализ граничных значений, техника тестирования всех пар и другие.

6.1.1. Метод эквивалентного разделения

Эквивалентное разделение заключается в том, что область всех возможных наборов **входных** данных программы по каждому параметру разбивают на конечное число групп – классов эквивалентности. Классы формируют так, чтобы все данные из одного класса были равнозначны с точки зрения результата работы программы (один и тот же результат). Для каждой переменной выделяют классы, содержащие допустимые и классы, содержащие недопустимые значения. Идентичность поведения системы для всех представителей класса, позволяет существенно сократить количество тестов, сэкономить время и улучшить структурированность тестирования.

Формирование классов эквивалентности является эвристическим процессом. Классы эквивалентности могут быть определены по спецификации ПП. Их выделяют, перебирая ограничения, установленные для каждого входного значения. Каждое ограничение разбивают на две или более групп, включающие допустимые и недопустимые значения. Например, если некоторый параметр x может принимать значения в интервале от 1 до 10, то для него выделяют один правильный класс $1 \le x \le 10$ и два неправильных: $x < 1$ и $x > 10$. Если в ходе работы программы, проверяется тип открываемого графического файла: bmp, jpg, или png, то определяют правильный класс эквивалентности **для каждого** значения и один неправильный класс, например, docx. **Правила** формирования классов эквивалентности сформулированы у Орлова [2].

Для каждого класса эквивалентности разрабатывается один тестовый вариант. Предполагается, что если набор-представитель класса обнаруживает некоторую ошибку, то и любой другой тестовый вариант этого класса эквивалентности тоже обнаружат эту ошибку. ***Например***, для диапазона допустимых значений от 1 до 10 создаем три тестовых варианта: 1) 5 – верное значение внутри интервала, 2) 0 (меньше 1) и 3) 22 (больше 10) – два неверных значения вне интервала.

Значения правильных классов для всех переменных проверяемой функции рекомендуется объединять в одном тесте. *Для каждого **неправильного** класса эквивалентности формируют **свой тест**.*

Техника эквивалентное разделение применяется к входным данным, результатам (обычно, для каждого правильного результата – свой класс, и для всех неправильных – один класс, но любой результат определяется входными данными), внутренним данным, данным с временной зависимостью. Использовать ее можно на всех уровнях тестирования.

6.1.2. Анализ граничных значений

Граничными называют значения на границах классов эквивалентности входных переменных или около них. Главная ***идея*** этой техники: точка изменения параметров поведения системы – одно из наиболее вероятных мест скопления ошибок. Практика показывает, что в этих местах программисты наиболее часто допускают ошибки. Тестовые варианты здесь создаются для проверки только ребер классов эквивалентности. При этом учитывают не только условия ввода, но и вывода. Например, если при проверке наличия окружности было записано $R \ge 0$ вместо $R > 0$, то задание граничного значения выявит ошибку: точка будет трактоваться как окружность.

Количество тестов для проверки граничных значений будет примерно в три раза больше количества границ. На каждой границе диапазона следует проверить по три значения: а) граничное значение; б) значение перед границей; в) значение после границы. ***Например,*** есть диапазон целых чисел, граница находится в

числе 10. Таким образом, будем проводить тесты с числом 9 (до границы), 10 (сама граница), 11 (после границы).

Анализ граничных значений может быть **применен** к полям, записям, файлам, числовым атрибутам нечисловых переменных (например, длина), циклам, структурам данных; физическим объектам (включая память); действиям с временной зависимостью и другим сущностям имеющим ограничения.

6.1.3. Анализ причинно-следственных связей

Метод использует алгебру логики. Причиной в данном случае называют отдельное входное условие или класс эквивалентности. Следствием – выходное условие или преобразование системы. Например, при проверке функции «добавить клиента», «Причина» – это ввод соответствующих значений в несколько полей («ФИО», «Адрес», и т.п.) экранной формы нажатие кнопки «Добавить». «Следствие» – результат этих действий: система добавляет клиента в базу данных и показывает его номер на экране (или не добавляет).

Идея метода – в соотнесении всех следствий к причинам т.е. в уточнении причинно-следственных связей. Метод позволяет получить высоко результативные тесты, выявить неполноту и неоднозначность спецификаций.

Построение тестов начинают с анализа спецификаций. Их разбивают на «рабочие» участки, выделяя в отдельные таблицы независимые группы причинно-следственных связей, определяют множество причин и следствий. На следующем этапе строят таблицу истинности, в которой каждой возможной комбинации причин ставится в соответствие следствие, вытекающее из смыслового содержания спецификации. При этом, принято обозначать: «1» – истина, «0» – ложь, «X» – безразличные состояния. Таблицу сопровождают примечаниями, задающими ограничения и описывающими комбинации причин и/или следствий, которые являются невозможными из-за каких-либо ограничений. Далее каждую строку таблицы преобразуют в тест.

6.1.4. Предугадывание ошибки

Метод основан на интуиции опыте тестировщика. Основная идея – перечислить в некотором списке возможные ошибки или ситуации, в которых они могут появиться и которые могут быть не учтены при проектировании тестов. На основе этого списка составляют тесты. Ниже приводится несколько идей для тестирования.

– *«Плохое» целое из «хороших» частей.* Данные, которые пользователь вводит в программу, часто состоят из нескольких элементов, образующих единое целое. Например, дата – из дня, месяца и года. Особенно на это стоит обратить внимание, если части целого задаются в отдельных элементах интерфейса (например, когда элементы управления расположены на небольшом экране).

– *Способ осуществления действия или точка входа.* Способ ввода данных (при подготовке тестов он обычно не конкретизируется) может быть важен для выявления еще некоторого количества ошибок. Стоит проверить каждый способ (с клавиатуры, с виртуальной клавиатуры, с помощью копирования и вставки, из контекстного меню, путем голосового ввода, перетаскивания) для каждой функции хотя бы по одному разу. Это удобно делать с помощью комбинаторных тестов, где способ осуществления той или иной операции будет выступать в качестве одного из параметров.

– *Поворот экрана в мобильных приложениях.* Каждый раз, когда пользователь поворачивает смартфон, система заново располагает элементы интерфейса на экране. Стоит проверить:

- хорошо ли выглядят все статические страницы приложения в вертикальном и в горизонтальном режиме;
- не теряются ли ранее введенные значения при повороте экрана;
- не прерывает ли поворот экрана во время действия процесс (проигрывание видео, скачивание файла и т.п.

– *Минимальный размер окна десктоп-приложения.* Стоит проверить, все ли элементы интерфейса доступны, можно ли ввести все данные во все поля, работают ли горячие клавиши и управление интерфейсом с помощью клавиатуры. Пользователь должен безболезненно взаимодействовать с интерфейсом, даже если размер окна минимален.

6.1.5. Техника тестирования всех пар (парное тестирование)

Это – метод тест-дизайна, основывающийся на тестировании комбинаций параметров и их значений. При тестировании нередко встречаются случаи, когда нужно проверить сочетания нескольких независимых факторов (например, версии ОС, браузера и других задействованных приложений). При этом проверка всех их комбинаций невозможна из-за комбинаторного возрастания их количества, а исключение каких-либо комбинаций рискованно. Данная техника позволяет существенно уменьшить общее количество тест-кейсов и сократить затраты на тестирование времени и средств.

Техника тестирования всех пар заключается в отборе всех комбинаций, покрывающих каждую пару факторов (а не все комбинации значений для всех переменных) и обеспечивающих, согласно известному закону Парето, 80% результата посредством 20 % действий. Она базируется на наблюдениях о том, что ошибки, вызванные взаимодействием трех и более факторов встречаются реже и менее критичны. Поэтому основное внимание следует уделить дефектам, которые проявляются при сочетании двух факторов.

Существуют два способа реализации техники тестирования всех пар: а) ортогональные матрицы и б) алгоритм перебора всех пар.

а) Подход **Ортогональные матрицы** (OrthogonalArrays) основывается на разработке двумерного массива комбинаций значений входных данных со следующими свойствами:

- в любых двух столбцах ортогональной матрицы встречаются **все комбинации** значений этих двух столбцов;
- если какая-то пара значений двух столбцов встречается несколько раз, то все возможные парные комбинации значений этих столбцов должны встретиться столько же раз.

Например, у нас есть 3 входных параметра, каждый из которых может принимать одно из двух значений (рис.6.1). Все возможные комбинации входных данных приведены в таблице 6.1. Переходим к построению ортогональной матрицы. Необходимо сделать так, чтобы два любые столбца содержали в себе все возможные комбинации только один раз. Для 1 и 3 параметров ортогональная матрица представлена в таблице 6.2 (значение в столбце 2 безразлично, выделено цветом). Для других пар – аналогично.

Параметр 1	Параметр 2	Параметр 3
Значение 1.1	Значение 2.1	Значение 3.1
Значение 1.2	Значение 2.2	Значение 3.2

Рисунок 6.1 – Значения входных параметров

Таблица 6.1- Комбинации входных данных

№ ком-бина-ции	Параметры		
	1	2	3
1	Значение1.1	Значение2.1	Значение3.1
2	Значение1.1	Значение2.1	Значение3.2
3	Значение1.1	Значение2.2	Значение3.1
4	Значение1.1	Значение2.2	Значение3.2
5	Значение1.2	Значение2.1	Значение3.1
6	Значение1.2	Значение2.1	Значение3.2
7	Значение1.2	Значение2.2	Значение3.1
8	Значение1.2	Значение2.2	Значение3.2

Таблица 6.2

№	Параметры		
	1	2	3
1	Знач.1.1	Знач.2.1	Знач.3.1
4	Знач.1.1	Знач.2.2	Знач.3.2
6	Знач.1.2	Знач.2.1	Знач.3.2
7	Знач.1.2	Знач.2.2	Знач.3.1

Итоговая ортогональная матрица представлена в таблице 5.3: результаты перебора первого параметра со вторым (строки 1–4), первого с третьим (строки 5–8) и второго с третьим (строки 9–12). Цветом выделены не обязательные значения. Заметим, что строки 1, 5, 9; 2, 11; 3, 7 и 6, 10 повторяются. Уберём из таблицы лишние комбинации (строки 5,7,9,10,11) и получим таблицу 5.4 из 7 тестовых вариантов.

Таблица 6.3

№	Параметры		
	1	2	3
1	Знач.1.1	Знач.2.1	Знач.3.1
2	Знач.1.1	Знач.2.2	Знач.3.1
3	Знач.1.2	Знач.2.1	Знач.3.1
4	Знач.1.2	Знач.2.2	Знач.3.1
5	Знач.1.1	Знач.2.1	Знач.3.1
6	Знач.1.1	Знач.2.1	Знач.3.2
7	Знач.1.2	Знач.2.1	Знач.3.1
8	Знач.1.2	Знач.2.1	Знач.3.2
9	Знач.1.1	Знач.2.1	Знач.3.1
10	Знач.1.1	Знач.2.2	Знач.3.2
11	Знач.1.1	Знач.2.1	Знач.3.1
12	Знач.1.1	Знач.2.2	Знач.3.2

Таблица 6.4

№	Параметры		
	1	2	3
1	Знач.1.1	Знач.2.1	Знач.3.1
2	Знач.1.1	Знач.2.2	Знач.3.1
3	Знач.1.2	Знач.2.1	Знач.3.1
4	Знач.1.2	Знач.2.2	Знач.3.1
5	Знач.1.1	Знач.2.1	Знач.3.2
6	Знач.1.2	Знач.2.1	Знач.3.2
7	Знач.1.1	Знач.2.2	Знач.3.2

Тесты можно оптимизировать, заменив не обязательные значения: добавим проверку пар из 5-7 строк во 2-ю и 3-ю строки. Итоговый результат (четыре теста) представлен на рисунке 6.2.

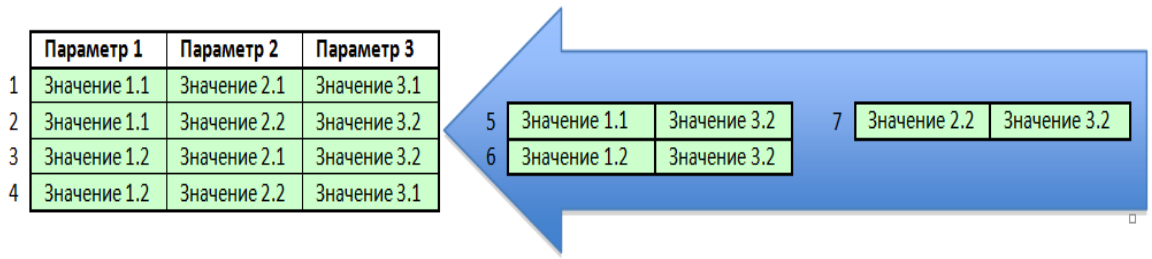

Рисунок 6.2 – Оптимизация по методу тестирования всех пар

Существуют специальные утилиты, позволяющие генерировать ортогональные массивы на основе входных данных и количества их возможных значений. Метод применим при различных видах тестирования: конфигурационном, регрессионном, нагрузочном, пользовательского интерфейса.

б) **Алгоритм перебора всех пар** (AllPairsAlgorithm)– Идея алгоритма состоит в выборе таких комбинаций переменных, чтобы существовали все возможные дискретные комбинации значений каждой пары входных переменных. При использовании алгоритма можно получить меньшее, по сравнению с методом ортогональных матриц, число комбинаций. **Например**, необходимо протестировать приложение для покупки/продажи «б/у-шных» ноутбуков. Исходные данные для тестирования приложения приведены в таблице 6.5

Таблица 6.5 – Исходные данные для тестирования приложения

Категория заказа	Местопо-ложение	Марка ноутбука		Операционная система (ОС)	Тип расчета	Тип доставки
покупка	Вологда	HP		доступна	наличный	почтой
продажа	Череповец	Lenovo		недоступна	безналичный	встреча
		Asus				

Переменные и значения систематизируем в виде таблицы 6.6 – 6.7. Столбцы необходимо организовать таким образом, чтобы первый имел наибольшее количество переменных (у нас это марка ноутбука), последний – наименьшее. Итак, в первом столбце записываем три значения Марки по два раза (два – это количество переменных следующего столбца, например, категория заказа). Для каждого набора в столбце 1 мы помещаем оба возможных значения столбца 2. Аналогично заполняем третий столбец.

Таблица 6.6 – Начало систематизации данных

Марка	Категория заказа	Место	ОС	Расчет	Доставка
HP	покупка	Вологда			
HP	продажа	Череповец			
Lenovo	покупка	Вологда			
Lenovo	продажа	Череповец			
Asus	покупка	Вологда			
Asus	продажа	Череповец			

Замечаем, что у нас есть комбинация покупка&Вологда и продажа& Череповец, но нет комбинации продажа&Вологда и покупка&Череповец. Исправим это, поменяв местами значения во втором наборе третьего столбца. Также заполняем 4-й и 5-й столбцы (таблица 6.7).

Таблица 6.7 – Продолжение систематизации данных

Марка	Категория заказа	Место	ОС	Расчет	Доставка
HP	покупка	Вологда	доступна	наличный	
HP	продажа	Череповец	недоступна	безналичный	
Lenovo	покупка	Череповец	доступна	безналичный	
Lenovo	продажа	Вологда	недоступна	наличный	
Asus	покупка	Вологда	недоступна	безналичный	
Asus	продажа	Череповец	доступна	наличный	

На колонке Доставка возникла проблема: не хватает строк для комбинаций покупка&встреча и продажа&почтой. Чтобы не нарушать отсортированные данные,

введем еще 2 строки для этих комбинаций. Значком тильды “~” отметим переменные, значение которых нам безразлично (таб.6.8).

Таблица 6.8 – Результат систематизации данных

Марка	Категория заказа	Место	ОС	Расчет	Доставка
HP	покупка	Вологда	доступна	наличный	почтой
HP	продажа	Череповец	недоступна	безналичный	встреча
~HP~	покупка	~Череповец~	~недоступна	~безналичный	встреча
Lenovo	покупка	Череповец	доступна	безналичный	почтой
Lenovo	продажа	Вологда	недоступна	наличный	встреча
~Lenovo~	продажа	~Вологда~	~доступна~	~наличный~	почтой
Asus	покупка	Вологда	недоступна	безналичный	почтой
Asus	продажа	Череповец	доступна	наличный	встреча

В результате мы получили 8 готовых тест-кейсов вместо 96 (чтобы протестировать все возможные комбинации, необходимо составить 2 x 2 x 3 x 2 x 2 x 2 = **96** тестовых наборов). Существуют специальные утилиты, позволяющие строить все пары переменных на AllPairsAlgorithm по таблице со всеми переменными и их значениями.

6.1.6. Техника «таблица принятия решений» (таблица решений)

Этот метод тест-дизайна основан на установлении связи между условиями и действиями. Метод позволяет компактно представить модели со сложной логикой, связи между множеством независимых условий и действий.

Таблица принятия решений, как правило, включает четыре квадранта:

Условия	Варианты выполнения условий
Действия	Необходимость действий

Например, так выглядит таблица решений для ситуации «неожиданно погас свет»:

Условия:	Варианты выполнения условий		
Свет в соседней комнате горит	Да	Нет	Нет
Свет у соседей горит	-	Да	Нет
Действия:	Необходимость действий		
Поменять лампочку	X		
Проверить пробки		X	
Позвонить электрику		X	X
Позвонить диспетчеру			X

Каждый столбик в правой части таблицы представляет собой тестовый вариант. Вариантов выполнения *условия* может быть несколько, действия могут быть элементарными или ссылаться на другие таблицы принятия решений, необходимость выполнения действий может оказаться упорядоченной. В более сложных таблицах может применяться нечёткая логика.

6.1.7. Тестирование состояния сущностей

Этот метод тест-дизайна базируется на схеме, в которой описаны все возможные пути перехода сущности из одного состояния в другое. В большинстве случаев требований к состояниям сущности в явном виде нет. Нужную информацию можно получить из сценариев использования системы.

Рассмотрим, например, сценарий «путь в университет» для сущности «студент». Будем считать, что студент может: Выйти из дома, Сесть в автобус, Выйти из автобуса, войти в Университет. Также введем ограничение, если студент вошёл в университет, то выйти из него он уже не может.

Тогда после выхода из дома у студента есть несколько вариантов развития событий: поехать в университет на автобусе, пойти пешком, вернутся пойти домой – 3 перехода состояния. Если он сел в автобус, то когда-нибудь из него выйдет. Добавляется ещё 1 переход – выйти из автобуса. Далее можно сесть в идущий в обратном направлении автобус, пойти домой пешком или всё-таки отправится в университет – итого 3 перехода состояния. В итоге получили семь состояний перехода, которые необходимо проверить. Из рисунка 6.3. видно, что эти семь переходов состояний можно покрыть четырьмя тестами. Это будут позитивные тесты, направленные на проверку валидных переходов состояний (действия, обозначенные стрелками). Стоит уделить внимание и некорректным переходам между состояниями сущностей (негативные тесты). Можно попробовать удалить то, что уже было удалено, отредактировать уже отправленное письмо и т.п. Во всех случаях ПП должен корректно обрабатывать нештатные ситуации.

Рисунок 6.3 – Схема переходов состояний

6.2. Стратегия "белого (прозрачного) ящика"

В основе стратегии «прозрачного (белого) ящика» лежит идея об открытости логики работы программы, которая может быть представлена в виде кода, алгоритма и т.п., и покрытие тестами всех операторов, всех возможных маршрутов, предусмотренных алгоритмом. Здесь ***маршрутом*** называют последовательность операторов, которые выполняются при данном конкретном варианте исходных данных. Считают, что программа проверена полностью, если с помощью тестов удается пройти **по всем возможным ее маршрутам**. Тестирование в этом случае называют структурным или тестированием по маршрутам.

Маршрут, который включает новый, по сравнению с ранее составленными, оператор обработки или новое условие называется *независимым.* Все независимые пути образуют базовое множество маршрутов алгоритма. Число независимых линейных путей (а значит и минимальное количество тестов, необходимое для покрытия маршрутов) в базовом множестве определяется цикломатической сложностью алгоритма.

Для вычисления цикломатической сложности необходимо на основе алгоритма (текста) программы построить **потоковый граф.** Граф состоит из вершин и дуг. Дуги (ориентированные ребра) отображают направление передачи управления между операторами (поток управления). Вершины (узлы) соответствуют линейным участкам программы и делятся на операторные и предикатные. Направление передачи управления операторным узлом всегда однозначно, поэтому из него выходит одна дуга. Направление передачи

управления предикатным узлом зависит от выполнения прописанного в нем условия, поэтому из него выходит две дуги. В потоковом графе предикатные узлы соответствуют ***простым*** *условиям,* поэтому каждое составное условие программы отображается в несколько предикатных узлов. Пример такого преобразования для фрагмента программы *if a OR b then X else Y end if* (где *a, b* – простые условия, *X, Y* – линейные участки программы*)* представлен на рисунке 6.4*.*

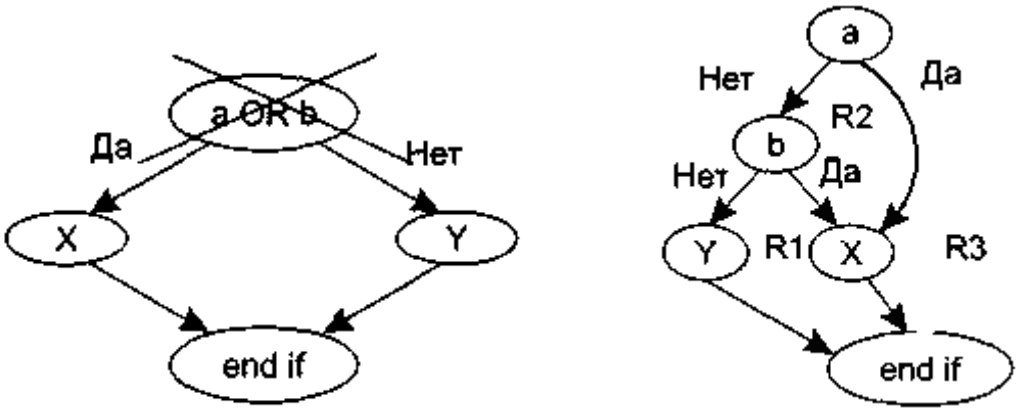

Рисунок 6.4 – Преобразование фрагмента программы в потоковый граф

Цикломатическая сложность алгоритма V(G) вычисляется одним из трех способов[2]:

1) по количеству регионов. Регион – замкнутая область, образованная дугами и узлами. Окружающая граф среда – дополнительный регион. Цикломатическая сложность равна количеству регионов потокового графа;

2) по количеству дуг и узлов потокового графа. Если обозначить E – количество дуг, N – количество узлов потокового графа, то

$$V(G) = E-N+2;$$

3)) по количеству предикатных узлов p:

$$V(G) = p +1.$$

Тесты, составленные на основе потокового графа и обеспечивающие проверку базового множества маршрутов алгоритма, гарантируют однократное выполнение каждого оператора и выполнение каждого условия по True- и по False-ветви. Порядок формирования независимых путей: от самого короткого к самому длинному.

Например, цикломатическая сложность алгоритма, представленного на рисунке 6.4, V(G) = 3:

1) V(G) = 3 региона: RI, R2, R3;
2) V(G) = 7 дуг – 6 узлов + 2 = 3;
3) V(G) = 2 предикатных узла + 1 = 3.

Независимые пути: Путь 1: a – x – end if.
Путь 2: a – b – x – end if.
Путь 3: a – b – y – end if.

Следовательно, потребуется три теста, которые должны обеспечить выполнение программы по этим маршрутам.

Однако число неповторяющихся маршрутов может быть очень велико. В этом случае формирование тестовых наборов для тестирования маршрутов может осуществляться **по другим критериям**: покрытие операторов, покрытие решений (переходов), покрытие условий, покрытие решений/условий, комбинаторное покрытие условий, покрытие потоков данных, покрытие циклов.

В этом случае для формирования тестов программу представляют в виде графа (граф-схемы), вершины которого соответствуют операторам программы, а дуги – возможным вариантам передачи управления. Линейные участки алгоритма можно представлять одной вершиной. Пример граф-схемы приведён на рис. 6.5, (справа). Обратите внимание, что в этом случае ***возле каждой дуги записывают условие прохождения по ней*** в явном виде и сложные условия на простые не разбивают. Не следует забывать и про преобразования переменных на линейных участках. Например, в узле 3 переменная x меняется и в узле 4 применяется уже новое её значение.

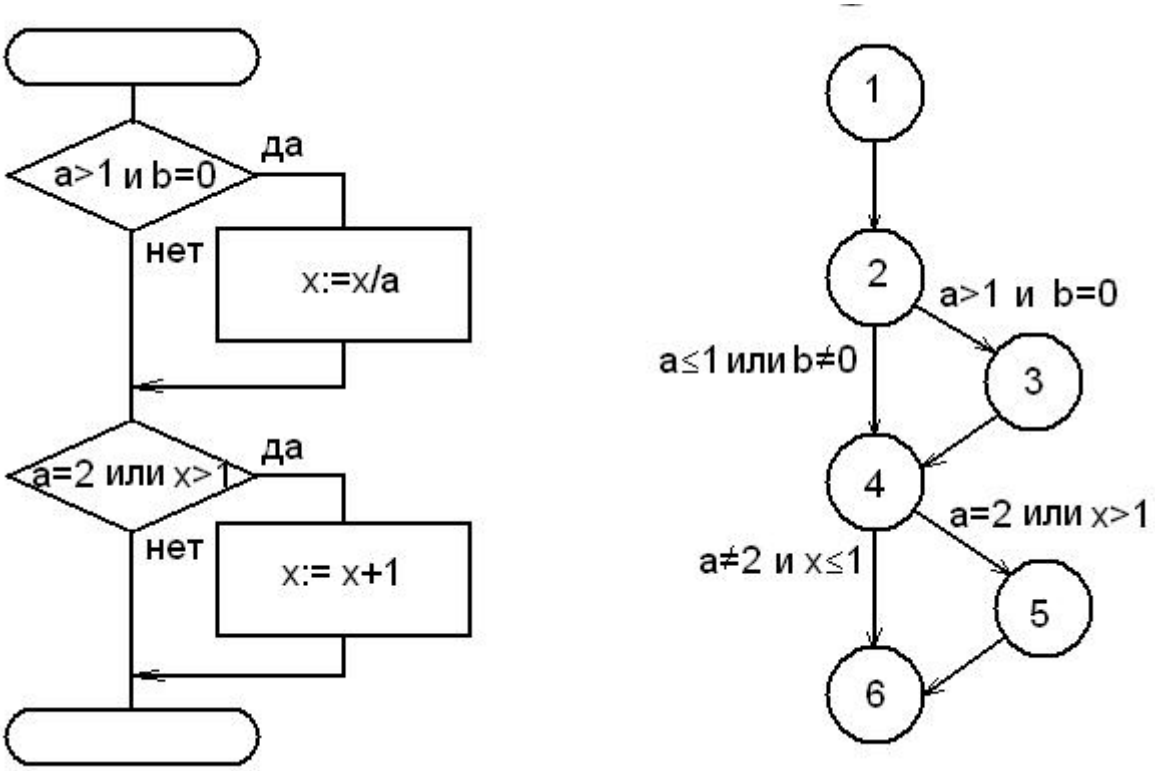

Рисунок 6.5 – Алгоритм (слева) и граф-схема (справа) процедуры

Критерий **покрытия операторов** подразумевает такой подбор тестов, чтобы каждый оператор программы выполнялся, минимум, один раз.

Для приведенного примера покрытие операторов будет реализовано посредством одного теста:

Исходные данные): a = 2, b = 0, x = 3;

Ожидаемые результаты: a = 2, b = 0, x = 2,5; проверен путь 1-2-3-4-5-6.

Всё просто. Однако, если в первом условии по ошибке записали «ИЛИ» вместо «И», а во втором условии x сравнивается не с единицей, данный тест эти дефекты выявить не сможет.

Более «сильным» по сравнению с критерием покрытия операторов является критерий **покрытие решений (переходов).** Для его реализации необходимо такое количество и состав тестов, чтобы обеспечить проверку каждого перехода в каждом предикатном узле (т.е результат каждого решения должен принять значения «истина» или «ложь», по крайней мере, по одному разу).

По этому критерию рассматриваемую программу (рис. 6.5) можно проверить двумя тестами, покрывающими либо пути: 1-2-4-6 и 1-2-3-4-5-6, либо пути: 1-2-3-4-6 и 1-2-4-5-6, например:

При a = 3, b = 0, x = 6 проверяем маршрут 1-2-3-4-5-6 (оба решения *(a>1)and(b=0) и (a=2)or(x>l)* будут истинны) ;
При a = -2, b = 1 , x = 1 – путь 1-2-4-6 (оба решения – ложны).

Критерий **покрытия условий** предполагает формирование некоторого количества тестов, достаточного для того, чтобы результат проверки каждого условия в решении принимал значения «истина» или «ложь», по крайней мере, один раз. В рассматриваемом фрагменте надо проверить четыре условия:

l) a>l; 2) b = 0; 3) a = 2; 4) x>1.

Тесты, удовлетворяющие этому условию:

a= 2, b = 0, x = 4 – путь 1-2-3-4-5-6, все четыре условия истинны;
a = 1, b = 1, x = 1 – путь 1-2-4-6, все четыре условия ложны.

Согласно критерию **покрытия решений/условий** тесты должны составляться так, чтобы, **по крайней мере, один раз** управление передавалось каждому оператору и выполнились все возможные результаты каждого условия и все результаты каждого решения.

Тесты, составленные для покрытия условий, этому требованию удовлетворяют: первый тест (путь 1-2-3-4-5-6) обеспечивает истинность всех четырёх условий и обоих решений, второй (путь 1-2-4-6) – их ложность. Но это случайное совпадение. Критерий покрытия условий часто удовлетворяет критерию покрытия решений, но не всегда.

Критерий **комбинаторное покрытие условий** требует создания такого множества тестов, чтобы выполнить все операторы и проверить все возможные комбинации условий в каждом решении минимум один раз.

В нашем фрагменте два решения по два условия в каждом. Следовательно, для реализации комбинаторного покрытия условий необходимо покрыть тестами восемь комбинаций:

1) a>1, b = 0; 2) a>1, b ≠ 0; 3) a≤1, b = 0; 4) a≤1, b ≠ 0
5) a = 2, x>1;6) a = 2, x≤1;7) a ≠ 2, x>1; 8) a ≠ 2, x≤1.

Эти комбинации можно проверить четырьмя тестами:

a = 2, b = 0, x = 4 – проверяет комбинации 1) и 5) (путь 1-2-3-4-5-6);
a = 2, b = 1, x = 1 – проверяет комбинации 2) и 6) (путь 1-2-4-5-6);
a= 1, b = 0, x = 2 –проверяет комбинации3) и 7) (путь 1-2-4-5-6);
a = 1 , b = 1 , x = 1 – проверяет комбинации 4) и 8) (путь 1-2-4-6).

То, что четыре теста проверяют все комбинации – случайность. Вполне могла возникнуть необходимость в реализации восьми тестов. Также можно заметить, что два теста покрывают один и тот же маршрут, и при этом путь 1-2-3-4-6 оказался пропущенным. Имеет смысл подготовить для этого маршрута отдельный тест. Некоторые комбинации вообще реализовать невозможно. Например, для условия *(a=2)or(a>100)* нельзя задать a, при котором оба условия будут истинны.

Покрытие области определения логических выражений. Кроме тестирования ветвей условий, стоит проверить области определения логических выражений. Выражение вида *E1 <оператор отношения> E2*, где E1 и E2 – арифметические выражения, а *<оператор отношения>* один из операторов: <, >, =, $\neq$, $\leq, \geq$, проверяется тремя тестами – значение E1 больше, чем E2, равно E2 и меньше, чем E2. Если оператор отношения неправилен, а E1 и E2 корректны, то эти три теста гарантируют обнаружение ошибки оператора отношения. Для определения ошибок в E1 и E2 следует задать значение E1 большим или меньшим, чем E2, обеспечив как можно меньшую разницу между этими значениями.

Тестовое покрытие циклов. При проверке циклов акцент делается на правильность их конструкций. Количество тестовых вариантов для проверки цикла по принципу «белого ящика» зависит от его типа: простой, вложенный (рис.6.6), объединенный, неструктурированный.

Рисунок 6.6 – Объемлющий и вложенный циклы

Например, для проверки ***простых*** циклов с количеством повторений n набор тестов может включать: 1) только один проход цикла; 2) два прохода цикла; 3) t проходов цикла, где t < n; 4) n–1, n, n+1 проходов цикла.

Тестирование ***вложенных*** циклов начинают с самого внутреннего цикла, установив при этом минимальные значения параметров всех остальных циклов. К внутреннему циклу применяют тесты простого цикла, добавив к ним тесты для исключенных значений и значений, выходящих за пределы рабочего диапазона. Далее тестируют следующий по порядку объемлющий цикл. При этом сохраняются минимальные значения параметров для всех внешних (объемлющих) циклов и типовые значения для всех вложенных циклов. Работа продолжается до тех пор, пока не будут протестированы все циклы.

Объединенные циклы тестируют как простые, если каждый из циклов независим от других, или как вложенные при наличии зависимости.

Неструктурированные циклы тестированию не подлежат.

Тестовое покрытие потоков данных. Тесты, составленные по этому критерию, используются при проверке информационной структуры программы: работу любой программы можно рассматривать как обработку и преобразование потока данных. В этом случае потоковый граф дополняется информационными связями. *Пример* потокового графа программы с управляющими (сплошные дуги) и информационными (пунктирные дуги) связями представлен на рисунке 6.7. Преобразование потока данных можно описать следующим образом:

— в вершине **1** определяются значения переменных **a, b**;
— значение переменной **a** используется в вершине **4**;
— значение переменной **b** используется в вершинах **3, 6**;
— в вершине **4** определяется значение переменной **c**, которая используется в вершине **6**.

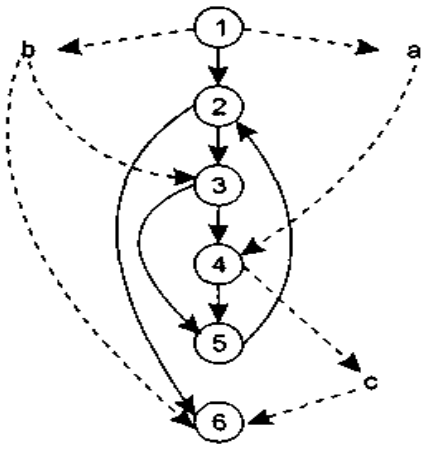

Рисунок 6.7 – Информационные связи потокового графа

Следующий шаг – выделение цепочек *определения-использования* данных. Так называют конструкцию [x, i, j], где x – имя переменной, которая определена в i-й вершине и используется в j-й вершине. В данном графе таких цепочек четыре: [a,1,4], [b, 1,3], [b, 1,6], [c, 4, 6]. Очевидно, что тестирование этих цепочек связано с тестированием маршрутов, при этом необходимо акцентировать внимание потоках данных.