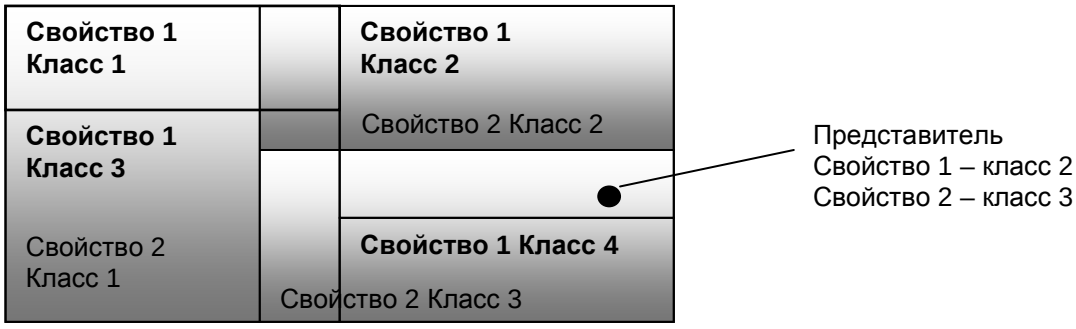
3. Модульное и интеграционное тестирование

Модульное и интеграционное тестирование тесно связаны с технологическим процессом конструирования ПО. По правилам хорошего тона тестовое покрытие модулей должно быть составной частью программного проекта, а по методологии экстремального программирования – его разработка должна быть выполнена одновременно с проектированием интерфейса модуля, до начала его содержательного наполнения.

Как уже было показано, исчерпывающее тестирование невозможно не только из-за многообразия наборов и последовательностей входных данных, так и из-за многообразия внешних факторов, способных вызвать ошибку. Следовательно, необходимо выбрать некоторое количество тестовых наборов, исполнение которых должно давать какую-то гарантию полноты тестирования – **тестовое покрытие**.

Выбор тестового покрытия производится на основе классификации тестовых наборов. Вводятся **классы эквивалентности** тестовых наборов **по значению какого-либо свойства программного кода, входных данных или внешних параметров**. Предполагается, что на тестах одного класса программа будет демонстрировать одинаковое поведение**.** Т.е. если один из наборов класса будет фиксировать ошибку, то все остальные также будут фиксировать ее, и наоборот. Тогда для каждого класса эквивалентности можно выбрать одного представителя, а из всех представителей сформировать тестовое покрытие.

Классы эквивалентностей для каждого свойства должны покрывать все множество тестовых наборов, а классы для разных свойств будут пересекаться. Т.е. один тестовый набор может одновременно являться представителем нескольких классов эквивалентности - по одному от значения каждого свойства.



Свойства, на основнии которых строятся классы, могут иметь отношение к компонентам программного кода, тогда тестирование называется **структурным,** и к ожидаемому поведению программы, тогда оно называется **функциональным.**

Структурное тестирование

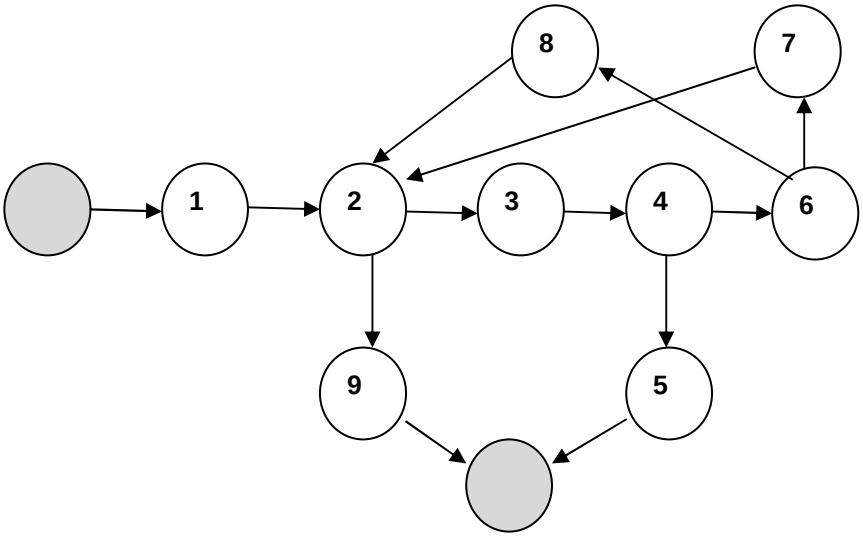
Исполнение прораммы можно рассматривать как поледовательность исполнения команд (операторов) – **поток управления** и последовательность преобразования данных – **поток данных**. Структурное тестирование в качестве критерия тестирования выбирает элементы или сочетания элементов потока команд (потока данных), а полноту тестирования оценивает через покрытие этих элементов (сочетаний) тестовым набором.

Наиболее распространенным является тестирование потока команд, т.к. синтаксис языков программирования на уровне отдельных модулей (функций, методов) описывает программный код как поток команд.

Структурное тестирование или тестирование потока управления в качестве критерия тестирования выбирает последовательности исполняемых операторов программного кода. Модель программы при структурном тестировании рассматривается в виде триады управляющих структур – действие (оператор) – условие – переход, т.е. фактически в методологии блок-схемы (или диаграммы состояний-переходов).

Поскольку для построения тестового покрытия важно не содержание элементов управления (его корректность как раз и проверяется тестами), а структура связей между ними (переходов по управлению), то вместо блок-схемы программы используется **упарвляющий граф программы,** в которой операторы представлены вершинами, а переходы между ними – дугами. Линейный участкок графа (линейная последовательность операторов) может быть «склеена» в одну вершину. Если условие является составным, то каждое элементарное условие должно быть представлено отдельной вершиной графа, а логические операции между ними отображены в виде дуг. В качестве примера рассмотрим управляющий граф программы двоичного поиска.

```
//-----------------------------------------------------46-01.cpp (cprog)
//------Двоичный поиск в упорядоченном массиве
 int binary(int c[], int n, int val){   // Возвращает индекс найденного
 int a,b,m;                             // Левая, правая границы и
 for(a=0,b=n-1; a <= b;) {              // середина – вершины 1,2
    m = (a + b)/2;                      // Середина интервала – вершина 3
    if (c[m] == val)                    // Значение найдено – вершина 4
          return m;                     // вернуть индекс найденного – вершина 5
     if (c[m] > val)                    // вершина 6
          b = m-1;                      // Выбрать левую половину – вершина 7
     else
          a = m+1;                      // Выбрать правую половину – вершина 8
     }
return -1; }                            // Значение не найдено – вершина 9
```



По управляющему графу можно ввести оценку структурной сложности программы – **цикломатическую сложность,** которая определяется как

S = количество ребер – количество вершин + 2

Для приведенной программы эта сложность с учетом начальной и конечной вершин состоявляет 4.

Критерии тестирования при структурном подходе основаны на различных требованиях к обязательному исполнению в тестовом покрытии элементов управляющих структур:

- Тестирование операторов – каждые оператор (действие) должен быть выполнен в покрытии хотя бы один раз;
- Тестирование условий – каждое условие в тестовом покрытии должно быть выполнено не менее 2 раз – по истинному и ложному срабатыванию;
- Тестирование путей (маршрутов) – каждая возможная последовательность действий и вариантов срабатывания условий должна быть выполнена один раз. Для управляющего графа – каждый возможный путь в графе должен быть выполнен один раз.

Отдельно следует обсудить **тестирование циклов.** Наличие обратной связи означает, что тестирование путей предполагает выполнение циклов с любым количеством шагов. Обычно ограничиваются качественным многообразием **- 0,1,2, m<n, n-1,n,n+1**, где **n –** стандартная число повторений цикла.

Достоинства и недостатки структурного тестирования:

- **(+)** структурное тестирование идеально подходит для тестирования логически сложных участков кода. Тестирование путей в этом случае фактически представляет собой проверку корректности всех результатов комбинаторного перебора вариантов срабатывания условий;
- **(+,-)** структурное тестирование имеет интегральный характер. Т.е. если в программе имеется ошибка **типа «опечатка»,** которая искажает большинство тестовых наборов и локализована в определенной точке программы, то структурное тестирование будет полезным. **Ошибки граничных условий**, связанные с отсутствием в программе реакции на граничные значения, вообще не будут выявлены, потому что в тестовом покрытии, построенном по структурному критерию, соответствующие наборы **никогда не встретятся** (не ищите в программе то, чего там нет).
- **(-)** если программа интенсивно использует внутренние данные для реализации собственной логики, то простая проверка исполнения всех операторов или условий недостаточна для тестирования логики ее работы. Для этого необходима проверка сочетаний различных условий (т.е. тестирование путей), которое в циклических программах ведет к комбинаторному росту количества вариантов тестирования и тестовых наборов;
- **(-)** для внутренних условий в программе не всегда могут быть легко определены тестовые наборы, определяюзщие те или иные значения этих условий. Например, в сортировке циклическим слиянием производятся сложные машипуляции групп элементов и для проверки всех условий в группах необходимо весьма искусно подбирать последовательность данных в исходном массиве;

Примечание. Диаграмма состояний-переходов относит нас к модели конечных автоматов, т.е. управляющих структур, не имеющих внутренней памяти. Поэтому в чистом виде структурный подход мало подходит для тестирования программ, интенсивно использующих внутренние данные и состояния для организации взаимодействия своих частей (повторение вышесказанного).

Функциональное тестирование

Функциональное тестирование или тестирование по принципу «черного ящика» предполагает, что тестовое покрытие разрабатывается исходя из предполагаемого функционала тестируемого кода, а не его конкретной реализации. Определение существенных свойств данных не является простой формальностью, а должно исходить из анализа возможного влияния различных значений данных, их последовательностей и сочетаний и на поведение программы. Классы создаются на основе анализа следующей информации:

- Синтаксическая и семантическая структура (формат) входных данных;
- Синтаксическая и семантическая структура (формат) выходных данных;
- Типичные ошибки программирования и возможности из появления в данном программном коде;
- Инспекция программного кода – поиск классов эквивалентности и граничных значений.

Эквивалентные разбиения и граничные значения

Каждое свойство входных или выходных данных, внешних параметров или характеристик функционала должно быть разбито на классы значений, на которых программа должна вести себя одинаково. Кроме значений, которые являются допустимыми, необходимы также классы ошибочных (недопустимых) значений. Например, если допустимы значения входного параметра в диапазоне **a…b**, то они составляют один класс эквивалентности, а два других класса составляют значения **<a** и **>b**. Т.е. тестовый набор должен включать 3 представителей со значениями в указанных диапазонах.

Эквивалентные разбиения используют в качестве тестируемого значения одно значение из допустимого диапазона. Но на практике значительное количество дефектов – это **дефекты граничных условий**, когда программа некорректно работает в одной из граничных ситуаций (на первом шаге цикла, при первой инициализации, при пустом множестве и т.п..). Поэтому метод эквивалентных разбиений усиливается путем добавления проверок **граничных значений, а также значений рядом с граничным.** Т.е. для допустимых значений в интервале a..b тестовый набор должен включать уже 9 значений **(<a-1, a-1,a,a+1, a< v < b, b-1,b,b+1, > b+1).**

Пример. Построение тестового покрытия для программы подсчета количества слов в строке. Для начала дадим точное определение того, что является словом с точки зрения программы: «слово – ненулевая последовательность любых символов, кроме пробела». Также определим понятие входной строки: «входная строка содержит любое количество строк, разделенных последовательностями пробелов ненулевой длины». Исходя из этих определений попробуем перечислим свойства строки такого формата и их возможные значения:

Свойство **– размерность строки**:

- Пустая строка
- Непустая строка

Свойство – **количество слов:**

- 0
- 1
- 2 более

Свойство – **количество пробелов между словами:**

- 1
- 2 и более

Свойство – **наличие пробелов перед первым словом:**

- отсутствуют
- имеются

Свойство – **наличие пробелов после последнего слова:**

- отсутствуют
- имеются

Свойство – **длина слова:**

- 1
- 2 и более

Сюда же следует добавить ряд граничных значений, которые хотя и являются сочетанием предыдущих, но могут быть описаны как свойство строки в целом:

- пустая строка;
- строка, состоящая целиком из пробелов;
- строка из одного слова, не содержащая пробелов.

Формат входных (выходных) данных

При функциональном тестировании еще одним очевидным источником разнообразия тестовых наборов является синтаксическая структура (формат) входных и выходных данных. Предыдущий пример фактически анализирует возможные вавианты формата входной строки.

Типичные ошибки программирования

Мелкие типичные ошибки программирования типа «опечатки» обычно выявляются на стадии отладки, т.к. приводят к неработоспособности программы на любых тестах. Тем не менее, многие из них могут оказаться непокрытыми тестовыми наборами, разработанными для других свойств.

Например, тестовые наборы для тестирования сортировок не учитывают свойство элементов – быть отрицательными, нулевыми или положительности. Между тем, ошибки инициализации заключаются обычно в присваивании значения 0 вместо требуемого по алгоритму. Кроме того, программист может при разработке исходить из того, что входные данные – положительные. Исходя из этого, не мешало бы включить проверочный тестовый набор, в котором хотя бы один элемент – отрицательный.

Инспекция программного кода - «полупрозрачный» черный ящик

Функциональное тестирование предполагает, что структура программного кода не берется в качестве основы для формирования тестового покрытия. Но это не значит, что в него категорически запрещено «заглядывать», дабы лукавый не попутал. Наоборот. Анализ алгоритма (и программы, его реализующей), позволяет выделить дополнительные **условия и их граничные значения,** которые зависят от особенностей реализации, но не отражены в основном функционале. Это можно рассматривать как добавление элементов структурного тестирования.

Пример. Циклическое слияние на массиве использует последовательности элементов длиной, кратной степени двойки. Если размерность массива не кратна этой степени, то возникают некоторые технологические особенности обработки «хвоста», которые могут

быть решены по-разному – от введения дополнительных условий, до расширения размерности вспомогательного массива. Очевидно, что дополнительное граничное условие для такой сортировки – кратность размерности степени двойки. Другой пример. **Пример. Однократное слияние** преобразует линейный массив в приблизительно квадратный. Опять-таки возникает «хвост» в виде не полностью заполненной последней строки. Дополнительное граничное условие – размерность массива, полный квадрат, например n=16.

Пример. Перераспределение памяти. Если в программе используется перераспределение памяти при достижении входными данными некоторой размерности, то в свойстве размерности должно появиться граничное значение «достаточно большой набор данных», который задействует указанный механизм.

Абсолютно черный ящик. В чистом виде функциональный подход является таким же однобоким, как и структурный. Покажем на примере. В любую сортировку внесем дефекты - отказ от сортировки при фиксированном значении размерности или элемента. Эти дефекты никакого отношения к функционалу не имеют, но они есть.

```
void sort(int a[], int n){
    if (n==1256) return;              // Не сортировать при размерности n=1256
    if (a[n-1]==1256) return;         // Не сортировать, если последний =1256
    }
```

Именно поэтому идеальным сочетанием может быть функциональный подход, дополненный структурным и инспекцией кода.

Распространенные свойства данных и алгоритмов и их граничные значения

Хотя свойств входных данных и алгоритмов их преобразования бесконечно много, имеется значительное число общих свойств и действий, которые характерны для многих случаев.

Свойство размерности входных данных. Входные данные (или их часть) представляют собой множество значений, размерность которого является свойством, определяющим граничные значения:

- Пустое множество;
- Единственный элемент;
- Два элемента;
- Более двух;
- Обычная (нормальная) размерность;
- «Очень большая размерность» - для определения дефектов масштабирования.

Размещение (включение) или удаление элемента. При включении элемента в последовательность возможны следующие граничные значения места включения:

- В начало;
- В конец;
- В середину;
- В пустую последовательность;
- В последовательность с единичным элементом.

Упорядочение (сортировка) набора данных.

- Исходный набор неупорядочена;
- Исходный набор упорядочен в прямом порядке;
- Исходный набор упорядочен в обратном порядке;
- В наборе имеются повторяющиеся элементы;
- В наборе имеются несколько групп повторяющихся элементов;
- Экстремальное значение находится в середине набора;

- Экстремальное значение находится в начале набора;
- Экстремальное значение находится в конце набора;
- В наборе несколько совпадающих экстремальных значений;

Поиск элемента (элементов) в наборе. При поиске элементов в наборе может быть определено несколько свойств найденного элемента с собственными наборами граничных значений.

Местоположение найденного в наборе:
- Найденный находится в начале набора;
- Найденный находится в конце набора;
- Найденный находится в середине набора;

Количество найденных:
- Не найден (отсутствует);
- Найден единственный;
- Найдено несколько (из них возвращен первый/последний/случайный);

Резюме. Идеальное тестовое покрытие.

Разумеется, идеального тестового покрытия не бывает. Но построение разумного покрытия, учитывающего по максимуму возможные дефекты – задача, к которой необходимо стремиться. Примерная стратегия может быть такой:

- Структурное тестирование операторов и условий – интегральная проверка кода.
- Функциональное тестовое покрытие на основе граничных значений входных/выходных данных и внешних параметров (черный ящик);
- Инспекция кода, определение внутренних параметров и их граничных значений – дополнительное функциональное покрытие («полупрозрачный» черный ящик);
- Инспекция кода – структурное тестовое покрытие для логически сложных участков кода.
- Инспекция кода – поиск потенциальных дефектов, мутационное тестирование для проверки полноты покрытия;
- Инспекция кода – предположения о типичных ошибках, дополнительные тестовые наборы.