**Лекция 2**

**Критерии выбора тестов**

**Основные вопросы**

1. Требования к идеальному критерию.
2. Классы критериев.
3. Структурные критерии.
4. Функциональные критерии.
5. Стохастические критерии.
6. Мутационный критерий.
7. Оценка покрытия программы и проекта.
8. Методика интегральной оценки тестированности.

### **Требования к идеальному критерию тестирования**

Требования к *идеальному критерию*:

1. **Критерий должен быть достаточным**, т.е. показывать, когда некоторое конечное множество тестов достаточно для тестирования данной программы.
2. **Критерий должен быть полным**, т.е. в случае ошибки должен существовать тест из множества тестов, удовлетворяющих критерию, который раскрывает ошибку.
3. **Критерий должен быть надежным**, т.е. любые два множества тестов, удовлетворяющих ему, одновременно должны раскрывать или не раскрывать ошибки программы
4. **Критерий должен быть легко проверяемым**, например вычисляемым на тестах

Для нетривиальных классов программ в общем случае **не существует полного и надежного критерия**, зависящего от программ или спецификаций.

Поэтому мы стремимся к идеальному общему критерию через реальные частные.

### Классы критериев

1. *Структурные критерии* используют информацию о структуре программы (критерии так называемого "белого ящика")
2. *Функциональные критерии* формулируются в описании требований к программному изделию ( **критерии так называемого "черного ящика"** )
3. Критерии *стохастического тестирования* формулируются в терминах проверки наличия заданных свойств у тестируемого приложения, средствами проверки некоторой статистической гипотезы.
4. *Мутационные критерии* ориентированы на проверку свойств программного изделия на основе подхода Монте-Карло.

### **Структурные критерии (класс I).**

*Структурные критерии* используют *модель программы* в виде "белого ящика", что предполагает *знание* исходного текста программы или *спецификации программы* в виде потокового графа управления. Структурная *информация* понятна и доступна разработчикам подсистем и модулей приложения, поэтому данный *класс* критериев часто используется на этапах модульного и *интеграционного тестирования* (*Unit testing*, *Integration testing*).

*Структурные критерии* базируются на основных элементах УГП, операторах, ветвях и путях.

* Условие критерия **тестирования команд** (критерий С0) - набор тестов в совокупности должен обеспечить прохождение каждой команды не менее одного раза. Это слабый критерий, он, как правило, используется в больших программных системах, где другие критерии применить невозможно.
* Условие критерия **тестирования ветвей** (критерий С1) - набор тестов в совокупности должен обеспечить прохождение каждой ветви не менее одного раза. Это достаточно сильный и при этом экономичный критерий, поскольку множество ветвей в тестируемом приложении конечно и не так уж велико. Данный критерий часто используется в системах *автоматизации тестирования*.
* Условие критерия **тестирования путей** (критерий С2) - набор тестов в совокупности должен обеспечить прохождение каждого пути не менее 1 раза. Если программа содержит цикл (в особенности с неявно заданным числом итераций), то число итераций ограничивается константой (часто - 2, или *числом классов* выходных путей).

*Структурные критерии* не проверяют соответствие спецификации, если оно не отражено в структуре программы. Поэтому при успешном тестировании программы по критерию *C2* мы можем не заметить ошибку, связанную с невыполнением некоторых условий спецификации требований.

### **Функциональные критерии (класс II)**

*Функциональный критерий* - важнейший для программной индустрии критерий тестирования. Он обеспечивает, прежде всего, *контроль* степени выполнения требований заказчика в программном продукте. Поскольку требования формулируются к продукту в целом, они отражают взаимодействие тестируемого приложения с окружением. При *функциональном тестировании* преимущественно используется модель "черного ящика". Проблема *функционального тестирования* - это, прежде всего, трудоемкость; дело в том, что документы, фиксирующие требования к программному изделию (*Software* requirement *specification*, *Functional specification* и т.п.), как правило, достаточно объемны, тем не менее, соответствующая проверка должна быть всеобъемлющей.

Ниже приведены частные виды *функциональных критериев*.

* **Тестирование пунктов спецификации** - набор тестов в совокупности должен обеспечить проверку каждого тестируемого пункта не менее одного раза.

Спецификация требований может содержать сотни и тысячи пунктов требований к программному продукту и каждое из этих требований при тестировании должно быть проверено в соответствии с критерием не менее чем одним тестом

* **Тестирование классов входных данных** - набор тестов в совокупности должен обеспечить проверку представителя каждого класса входных данных не менее одного раза.

При создании тестов классы входных данных сопоставляются с режимами использования тестируемого компонента или подсистемы приложения, что заметно сокращает варианты перебора, учитываемые при разработке тестовых наборов. Следует заметить, что перебирая в соответствии с критерием величины входных переменных (например, различные файлы - источники входных данных), мы вынуждены применять мощные тестовые наборы. Действительно, наряду с ограничениями на величины входных данных, существуют ограничения на величины входных данных во всевозможных комбинациях, в том числе проверка реакций системы на появление ошибок в значениях или структурах входных данных. Учет этого многообразия - процесс трудоемкий, что создает сложности для применения критерия

* **Тестирование правил** - набор тестов в совокупности должен обеспечить проверку каждого правила, если входные и выходные значения описываются набором правил некоторой грамматики.

Следует заметить, что грамматика должна быть достаточно простой, чтобы трудоемкость разработки соответствующего набора тестов была реальной (вписывалась в сроки и штат специалистов, выделенных для реализации *фазы тестирования*)

* **Тестирование классов выходных данных** - набор тестов в совокупности должен обеспечить проверку представителя каждого выходного класса, при условии, что выходные результаты заранее расклассифицированы, причем отдельные классы результатов учитывают, в том числе, ограничения на ресурсы или на время (time out).

При создании тестов классы выходных данных сопоставляются с режимами использования тестируемого компонента или подсистемы, что заметно сокращает варианты перебора, учитываемые при разработке тестовых наборов.

* **Тестирование функций** - набор тестов в совокупности должен обеспечить проверку каждого действия, реализуемого тестируемым модулем, не менее одного раза.

Очень популярный на практике критерий, который, однако, не обеспечивает покрытия части функциональности тестируемого компонента, связанной со структурными и поведенческими свойствами, описание которых не сосредоточено в отдельных функциях (т.е. описание рассредоточено по компоненту).

Критерий тестирования функций объединяет отчасти особенности *структурных* и *функциональных критериев*. Он базируется на модели "полупрозрачного ящика", где явно указаны не только входы и выходы тестируемого компонента, но также состав и структура используемых методов (функций, процедур) и классов.

* **Комбинированные критерии для программ и спецификаций** - набор тестов в совокупности должен обеспечить проверку всех комбинаций непротиворечивых условий программ и спецификаций не менее одного раза.

При этом все комбинации непротиворечивых условий надо подтвердить, а условия противоречий следует обнаружить и ликвидировать.

### **Стохастические критерии (класс III)**

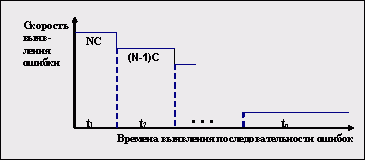
*Стохастическое тестирование* применяется при тестировании сложных программных комплексов - когда набор детерминированных тестов (X,Y) имеет громадную *мощность*. В случаях, когда подобный набор невозможно разработать и исполнить на *фазе тестирования*, можно применить следующую методику.

* Разработать программы - имитаторы случайных последовательностей входных сигналов {x}.
* Вычислить независимым способом значения {y} для соответствующих входных сигналов {x} и получить тестовый набор (X,Y).
* Протестировать приложение на тестовом наборе (X,Y), используя два способа контроля результатов:
  + Детерминированный контроль - проверка соответствия вычисленного значения yв\in \{ y\} значению y, полученному в результате *прогона теста* на наборе {x} - случайной последовательности входных сигналов, сгенерированной имитатором.
  + Стохастический контроль - проверка соответствия множества значений {yв}, полученного в результате прогона тестов на наборе входных значений {x}, заранее известному распределению результатов F(Y).

В этом случае множество Y неизвестно (его вычисление невозможно), но известен закон распределения данного множества.

Критерии *стохастического тестирования*

* **Cтатистические методы** окончания тестирования - стохастические методы принятия решений о совпадении гипотез о распределении случайных величин. К ним принадлежат широко известные: метод Стьюдента ( St ), метод Хи-квадрат ( \chi ^{2} ) и т.п.
* Метод *оценки скорости выявления ошибок* - основан на модели скорости выявления ошибок [[ 12 ]](https://www.intuit.ru/studies/courses/48/48/literature#literature.12), согласно которой тестирование прекращается, если оцененный интервал времени между текущей ошибкой и следующей слишком велик для *фазы тестирования* приложения.

**Рис. 3.1.**Зависимость скорости выявления ошибок от времени выявления

При формализации модели скорости выявления ошибок ([рис. 3.1](https://www.intuit.ru/studies/courses/48/48/lecture/1428?page=3#image.3.1)) использовались следующие обозначения:

N - исходное число ошибок в программном комплексе перед тестированием,

C - константа снижения скорости выявления ошибок за счет нахождения очередной ошибки,

t1, t2,… tn - *кортеж* возрастающих интервалов обнаружения последовательности из n ошибок,

T - время выявления n ошибок.

Если допустить, что за время T выявлено n ошибок, то справедливо соотношение (1), утверждающее, что *произведение* скорости выявления i ошибки и времени выявления i ошибки есть 1 по определению:

(1) (N-i+1)\*C\*ti = 1

В этом предположении справедливо соотношение (2) для n ошибок:

(2) N*C*t_{1}+(N-1)*C*t_{2}+…+(N-n+1)*C*t_{n}=n 
\\
    N*C*(t_{1}+t_{2}+…+t_{n}) - C*\Sigma (i-1)t_{i} = n
\\
    NCT - C*\Sigma (i-1)t_{i} = n

Если из (1) определить ti и просуммировать от 1 до n, то придем к соотношению (3) для времени T выявления n ошибок


(3) \Sigma 1/(N-i+1) = TC

Если из (2) выразить C, приходим к соотношению (4):

(4) C = n/(NT - \Sigma (i-1)t_{i})

Наконец, подставляя C в (3), получаем окончательное соотношение (5), удобное для оценок:

(5) \Sigma 1/(N-i+1) = n/(N - 1/T*\Sigma (i-1)t_{i})

Если оценить величину N приблизительно, используя известные методы оценки числа ошибок в программе или данные о *плотности ошибок* для проектов рассматриваемого класса из исторической *базы данных* проектов, и, кроме того, использовать текущие данные об *интервалах между ошибками* t1, t2 … tn, полученные на *фазе тестирования*, то, подставляя эти данные в (5), можно получить оценку tn+1 -временного интервала необходимого для нахождения и исправления очередной ошибки (будущей ошибки).

Если tn+1>Td - допустимого времени тестирования проекта, то тестирование заканчиваем, в противном случае продолжаем *поиск* ошибок.

Наблюдая последовательность интервалов ошибок t1, t2 … tn, и время, потраченное на выявление n ошибок T=\Sigma t_{i}, можно прогнозировать *интервал* времени до следующей ошибки и уточнять в соответствии с (4) величину C.

Критерий Moranda очень практичен, так как опирается на информацию, традиционно собираемую в процессе тестирования.

### **Мутационный критерий (класс IV).**

Постулируется, что профессиональные программисты пишут сразу почти правильные программы, отличающиеся от правильных мелкими ошибками или описками типа - *перестановка* местами максимальных значений индексов в описании массивов, ошибки в знаках арифметических операций, занижение или завышение границы *цикла* на 1 и т.п. Предлагается подход, позволяющий на основе мелких ошибок оценить общее число ошибок, оставшихся в программе.

Подход базируется на следующих понятиях:

***Мутации*** - мелкие ошибки в программе.

***Мутанты*** - программы, отличающиеся друг от друга *мутациями* .

**Метод мутационного тестирования** - в разрабатываемую программу P вносят *мутации*, т.е. искусственно создают программы-*мутанты* P1, P2... Затем *программа* P и ее *мутанты* тестируются на одном и том же наборе тестов (X,Y).

Если на наборе (X,Y) подтверждается *правильность программы* P и, кроме того, выявляются все внесенные в программы-*мутанты* ошибки, то **набор тестов (X,Y) соответствует***мутационному критерию*, а тестируемая *программа* объявляется **правильной**.

Если некоторые *мутанты* не выявили всех *мутаций*, то надо расширять набор тестов (X,Y) и продолжать тестирование.

### **Оценка Покрытия Программы и Проекта**

*Тестирование программы* Р по некоторому критерию С означает покрытие *множества* компонентов программы P М = {m1...mk} по элементам или по связям

T = {t1...tn} - *кортеж* неизбыточных тестов ti.

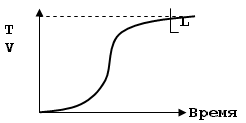
Тест ti неизбыточен, если существует покрытый им *компонент* mi из M(P,C), не покрытый ни одним из предыдущих тестов t1...ti-1. Каждому ti соответствует неизбыточный *путь* pi - последовательность вершин от входа до выхода.

V(P,C) - *сложность тестирования* Р по критерию С - измеряется *max* числом неизбыточных тестов, покрывающих все элементы *множества* M(P,C)

*DV*(P,C,Т) - остаточная *сложность тестирования* Р по критерию С - измеряется *max* числом неизбыточных тестов, покрывающих элементы *множества* M(P,C), оставшиеся непокрытыми, после прогона набора тестов Т. Величина *DV* строго и монотонно убывает от V до 0.

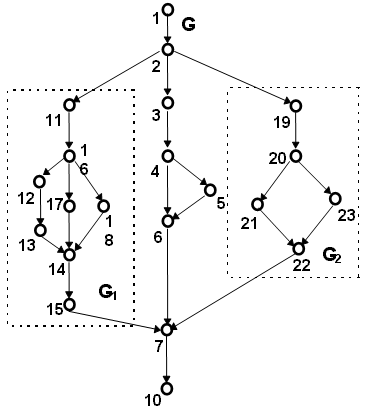
TV(P,C,Т) = (V-*DV*)/V - оценка *степени тестированности* Р по критерию С.

Критерий окончания тестирования TV(P,C,Т) >= L, где (0 <= L <= 1). L - уровень оттестированности, заданный в требованиях к программному продукту.

**Рис. 4.1.**Метрика оттестированности приложения

Рассмотрим две модели программного обеспечения, используемые при оценке оттестированности.

Для оценки *степени оттестированности* часто используется *УГП* - *управляющий граф программы*. *УГП* многокомпонентного объекта G ([Рис. 4.2](https://www.intuit.ru/studies/courses/48/48/lecture/1430?page=1#image.4.2), [Пример 4.4](https://www.intuit.ru/studies/courses/48/48/lecture/1430?page=2#example.4.4)), содержит внутри себя два компонента G1 и G2, *УГП* которых раскрыты.

**Рис. 4.2.**Плоская модель УГП компонента G

В результате *УГП* компонента G имеет такой вид, как если бы компоненты G1 и G2 в его структуре специально не выделялись, а *УГП* компонентов G1 и G2 были вставлены в *УГП* G. Для тестирования компонента G в соответствии с критерием путей потребуется прогнать *тестовый набор*, покрывающий следующий набор трасс *графа* G ([Пример 4.1](https://www.intuit.ru/studies/courses/48/48/lecture/1430?page=1#example.4.1)):

Пример 4.1. Набор трасс, необходимых для покрытия плоской модели УГП компонента G

P1(G) = 1-2-3-4-5-6-7-10;

P2(G) = 1-2-3-4-6-7-10;

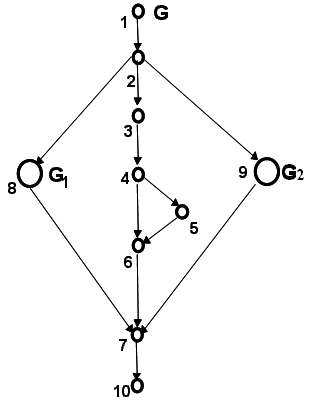
P3(G) = 1-2-11-16-18-14-15-7-10;

P4(G) = 1-2-11-16-17-14-15-7-10;

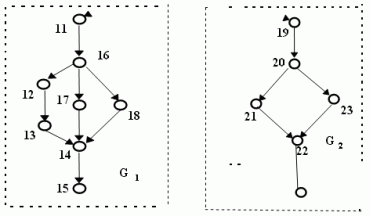
P5(G) = 1-2-11-16-12-13-14-15-7-10;

P6(G) = 1-2-19-20-23-22-7-10;

P7(G) = 1-2-19-20-21-22-7-10;

**Рис. 4.3.**Иерархическая модель УГП компонента G

*УГП* компонента G, представленный в виде иерархической модели, приведен на [Рис. 4.3](https://www.intuit.ru/studies/courses/48/48/lecture/1430?page=1#image.4.3), [Пример 4.5](https://www.intuit.ru/studies/courses/48/48/lecture/1430?page=2#example.4.5). В иерархическом *УГП* G входящие в его состав компоненты представлены ссылками на свои *УГП* G1 и G2 ([Рис. 4.4](https://www.intuit.ru/studies/courses/48/48/lecture/1430?page=1#image.4.4), [Пример 4.5](https://www.intuit.ru/studies/courses/48/48/lecture/1430?page=2#example.4.5))

**Рис. 4.4.**Иерархическая модель: УГП компонент G1 и G2

Для исчерпывающего тестирования иерархической модели компонента G в соответствии с критерием путей требуется прогнать следующий набор трасс ([Пример 4.2](https://www.intuit.ru/studies/courses/48/48/lecture/1430?page=1#example.4.2)):

4.2. Набор трасс, необходимых для покрытия иерархической модели УГП компонента G

P1(G) = 1-2-3-4-5-6-7-10;

P2(G) = 1-2-3-4-6-7-10;

P3(G) = 1-2-8-7-10;

P4(G) = 1-2-9-7-10.

Приведенный набор трасс достаточен при условии, что компоненты G1 и G2 в свою *очередь* исчерпывающе протестированы. Чтобы обеспечить выполнение этого условия в соответствии с критерием путей, надо прогнать все трассы [Пример 4.3](https://www.intuit.ru/studies/courses/48/48/lecture/1430?page=1#example.4.3).

4.3. Набор трасс иерархической модели УГП, необходимых для покрытия УГП компонентов G1 и G2

P11(G1)=11-16-12-13-14-15;

P12(G1)=11-16-17-14-15;

P13(G1)=19-20-23-22;

P21(G2)=19-20-21-22;

P22(G2)=11-16-18-14-15.

Оценка *степени тестированности* *плоской модели* определяется долей прогнанных трасс из набора необходимых для покрытия в соответствии с критерием С.

(1) TV(G,С) = (V-DV)/V = \sum PT_{i}(G) / (\sum P_{i}(G)),

где PTi(G) - тестовый *путь* ( ti ) в *графе* G *плоской модели* равен 1, если он протестирован (прогнан), или 0, если нет.

Например, если в *УГП* ([Пример 4.1](https://www.intuit.ru/studies/courses/48/48/lecture/1430?page=1#example.4.1)) тесты t6 и t7, которым соответствуют трассы P6 и P8, не прогнаны, то в соответствии с соотношением (1) для TV(G,С) *степень тестированности* будет оценена в 0.71.

Оценка тестированности иерархической модели определяется на основе учета оценок тестированности компонентов. Если трасса некоторого теста tj*УГП* G включает узлы, представляющие компоненты Gj1,..Gjm, оценка TV *степени тестированности* которых известна, то оценка тестированности PTi(G) при реализации этой трассы определяется не 1, а минимальной из оценок TV для компонентов.

Интегральная оценка определяется соотношением (2):

(2) TV(G,C) = (V-DV)/V = (\sum PT_{i}(G) * \sum (TV(G_{ij},C))) / (\sum P_{i}(G))

где PTi(G) - тестовый *путь* ( ti ) в *графе* G равен 1, если протестирован, или 0, если нет. В *путь* PTi*графа* G может входить j узлов модулей Gij со своей *степенью тестированности* TV(Gij,С), из которых мы берем *min*, что дает худшую оценку степени тестированности пути.

### Методика интегральной оценки тестированности

1. Выбор критерия С и приемочной оценки тестированности программного проекта - L
2. Построение древа классов проекта и построение *УГП* для каждого модуля
3. Модульное тестирование и оценка TV на модульном уровне
4. Построение *УГП*, интегрирующего модули в единую иерархическую ( *классовую* ) модель проекта
5. Выбор тестовых путей для проведения интеграционного или системного тестирования
6. Генерация тестов, покрывающих тестовые пути шага 5
7. Интегральная оценка тестированности проекта с учетом оценок тестированности модулей-компонентов
8. Повторение шагов 5-7 до достижения заданного уровня тестированности L