**Лекция 3**

**Разновидности тестирования**

**Основные вопросы**

1. Модульное тестирование.
2. Особенности интеграционного тестирования для объектно-ориентированного программирования.
3. Системное тестирование.
4. Регрессионное тестирование.
5. Комбинирование уровней тестирования

### **Модульное**

***Модульное тестирование*** - это тестирование программы на уровне отдельно взятых модулей, функций или классов. Цель *модульного тестирования* состоит в выявлении локализованных в модуле ошибок в реализации алгоритмов, а также в определении степени готовности системы к переходу на следующий уровень разработки и тестирования. *Модульное тестирование* проводится по принципу "белого ящика", то есть основывается на знании внутренней структуры программы, и часто включает те или иные методы анализа покрытия кода.

*Модульное тестирование* обычно подразумевает создание вокруг каждого модуля определенной среды, включающей *заглушки* для всех интерфейсов тестируемого модуля. Некоторые из них могут использоваться для подачи входных значений, другие для анализа результатов, присутствие третьих может быть продиктовано требованиями, накладываемыми компилятором и сборщиком.

На уровне *модульного тестирования* проще всего обнаружить дефекты, связанные с алгоритмическими ошибками и ошибками кодирования алгоритмов, типа работы с условиями и счетчиками циклов, а также с использованием локальных переменных и ресурсов. Ошибки, связанные с неверной трактовкой данных, некорректной реализацией интерфейсов, совместимостью, производительностью и т.п. обычно пропускаются на уровне *модульного тестирования* и выявляются на более поздних стадиях тестирования.

Именно эффективность обнаружения тех или иных типов дефектов должна определять стратегию *модульного тестирования*, то есть расстановку акцентов при определении набора входных значений. У организации, занимающейся разработкой программного обеспечения, как правило, имеется историческая *база данных* ( **Repository** ) разработок, хранящая конкретные сведения о разработке предыдущих проектов: о версиях и сборках кода ( **build** ) зафиксированных в процессе разработки продукта, о принятых решениях, допущенных просчетах, ошибках, успехах и т.п. Проведя *анализ* характеристик прежних проектов, подобных заказанному организации, можно предохранить новую разработку от старых ошибок, например, определив типы дефектов, *поиск* которых наиболее эффективен на различных этапах тестирования.

В данном случае анализируется этап *модульного тестирования*. Если *анализ* не дал нужной информации, например, в случае проектов, в которых соответствующие данные не собирались, то основным правилом становится *поиск* локальных дефектов, у которых код, ресурсы и *информация*, вовлеченные в дефект, характерны именно для данного модуля. В этом случае на *модульном уровне* ошибки, связанные, например, с неверным порядком или форматом параметров модуля, могут быть пропущены, поскольку они вовлекают информацию, затрагивающую другие модули (а именно, спецификацию интерфейса), в то время как ошибки в алгоритме обработки параметров довольно легко обнаруживаются.

Являясь по способу исполнения структурным тестированием или тестированием "белого ящика", *модульное тестирование* характеризуется степенью, в которой тесты выполняют или покрывают логику программы (исходный текст). Тесты, связанные со структурным тестированием, строятся по следующим принципам:

* На основе анализа *потока управления*. В этом случае элементы, которые должны быть покрыты при прохождении тестов, определяются на основе *структурных критериев* тестирования С0, С1,С2. К ним относятся вершины, дуги, пути *управляющего графа* программы (УГП), условия, комбинации условий и т. п.
* На основе анализа *потока данных*, когда элементы, которые должны быть покрыты, определяются при помощи *потока данных*, т. е. *информационного графа* программы.

**Тестирование на основе потока управления**. Особенности использования *структурных критериев* тестирования С0,С1,С2 были рассмотрены в лекции 3. К ним следует добавить критерий покрытия условий, заключающийся в покрытии всех логических (булевских) условий в программе. Критерии покрытия решений (ветвей - С1) и условий не заменяют друг друга, поэтому на практике используется комбинированный критерий покрытия условий/решений, совмещающий требования по покрытию и решений, и условий.

К популярным критериям относятся критерий покрытия функций программы, согласно которому каждая *функция* программы должна быть вызвана хотя бы один раз, и критерий покрытия вызовов, согласно которому каждый вызов каждой функции в программе должен быть осуществлен хотя бы один раз. Критерий покрытия вызовов известен также как критерий покрытия пар вызовов (*call* *pair* *coverage*).

**Тестирование на основе потока данных**. Этот вид тестирования направлен на выявление ссылок на неинициализированные переменные и избыточные присваивания (аномалий *потока данных*). Как основа для стратегии тестирования *поток данных*. Предложенная там стратегия требовала тестирования всех взаимосвязей, включающих в себя ссылку (использование) и *определение* переменной, на которую указывает *ссылка* (т. е. требуется покрытие дуг *информационного графа* программы). Недостаток стратегии в том, что она не включает критерий С1, и не гарантирует покрытия решений.

Стратегия требуемых пар также тестирует упомянутые взаимосвязи. Использование переменной в предикате дублируется в соответствии с числом выходов решения, и каждая из таких требуемых взаимосвязей должна быть протестирована. К популярным критериям принадлежит критерий СР, заключающийся в покрытии всех таких пар дуг v и w, что из дуги v достижима *дуга* w, поскольку именно на дуге может произойти потеря значения переменной, которая в дальнейшем уже не должна использоваться. Для "покрытия" еще одного популярного критерия Cdu достаточно тестировать пары (*вершина*, *дуга*), поскольку *определение* переменной происходит в вершине УГП, а ее использование - на дугах, исходящих из решений, или в вычислительных вершинах.

**Методы проектирования тестовых путей для достижения заданной степени тестированности в структурном тестировании**. Процесс построения набора тестов при структурном тестировании принято делить на три фазы:

* Конструирование УГП.
* Выбор тестовых путей.
* Генерация тестов, соответствующих тестовым путям.

Первая фаза соответствует статическому анализу программы, задача которого состоит в получении графа программы и зависящего от него и от критерия тестирования *множества* элементов, которые необходимо покрыть тестами.

На третьей фазе по известным путям тестирования осуществляется *поиск* подходящих тестов, реализующих прохождение этих путей.

Вторая фаза обеспечивает выбор тестовых путей. Выделяют три подхода к построению тестовых путей:

* *Статические методы*.
* *Динамические методы*.
* Методы реализуемых путей.

**Статические методы**. Самое простое и легко реализуемое решение - построение каждого пути посредством постепенного его удлинения за счет добавления дуг, пока не будет достигнута выходная *вершина* *управляющего графа* программы. Эта идея может быть усилена в так называемых адаптивных методах, которые каждый раз добавляют только один тестовый *путь* (*входной* тест), используя предыдущие пути (тесты) как руководство для выбора последующих путей в соответствии с некоторой стратегией. Чаще всего адаптивные стратегии применяются по отношению к критерию С1. Основной недостаток *статических методов* заключается в том, что не учитывается возможная нереализуемость *построенных путей* тестирования.

**Динамические методы**. Такие методы предполагают построение полной системы тестов, удовлетворяющих заданному критерию, путем одновременного решения задачи построения покрывающего множества путей и тестовых данных. При этом можно автоматически учитывать реализуемость или нереализуемость ранее рассмотренных путей или их частей. Основной идеей *динамических методов* является подсоединение к начальным реализуемым отрезкам путей дальнейших их частей так, чтобы: 1) не терять при этом реализуемости вновь полученных путей; 2) покрыть требуемые элементы структуры программы.

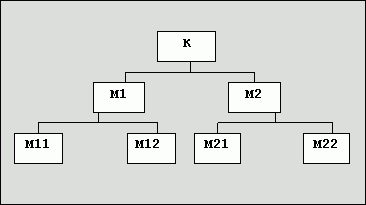
**Методы реализуемых путей**. Данная методика заключается в выделении из *множества* путей подмножества всех реализуемых путей. После чего покрывающее множество путей строится из полученного подмножества реализуемых путей.

Достоинство *статических методов* состоит в сравнительно небольшом количестве необходимых ресурсов, как при использовании, так и при разработке. Однако их реализация может содержать непредсказуемый *процент* брака (нереализуемых путей). Кроме того, в этих системах переход от покрывающего *множества* путей к полной системе тестов *пользователь* должен осуществить вручную, а эта работа достаточно трудоемкая. *Динамические методы* требуют значительно больших ресурсов как при разработке, так и при эксплуатации, однако увеличение затрат происходит, в основном, за счет разработки и эксплуатации аппарата определения реализуемости пути (символический *интерпретатор*, *решатель* неравенств). Достоинство этих методов заключается в том, что их продукция имеет некоторый качественный уровень - реализуемость путей. Методы реализуемых путей дают самый лучший результат.

### Интеграционное тестирование

***Интеграционное тестирование*** - это тестирование части системы, состоящей из двух и более модулей. Основная задача *интеграционного тестирования* - поиск дефектов, связанных с ошибками в реализации и интерпретации интерфейсного взаимодействия между модулями.

С технологической точки зрения *интеграционное тестирование* является количественным развитием *модульного*, поскольку так же, как и *модульное тестирование*, оперирует интерфейсами модулей и подсистем и требует создания тестового окружения, включая *заглушки* ( **Stub** ) на месте отсутствующих модулей. Основная разница между *модульным* и *интеграционным тестированием* состоит в целях, то есть в типах обнаруживаемых дефектов, которые, в свою *очередь*, определяют стратегию выбора входных данных и методов анализа. В частности, на уровне *интеграционного тестирования* часто применяются методы, связанные с покрытием интерфейсов, например, вызовов функций или методов, или *анализ* использования интерфейсных объектов, таких как глобальные ресурсы, средства коммуникаций, предоставляемых операционной системой.

**Рис. 5.1.**Пример структуры комплекса программ

На [Рис. 5.1](https://www.intuit.ru/studies/courses/48/48/lecture/1432?page=2#image.5.1) приведена структура комплекса программ K, состоящего из оттестированных на этапе *модульного тестирования* модулей M1, M2, M11, M12, M21, M22. Задача, решаемая методом *интеграционного тестирования*, - тестирование межмодульных связей, реализующихся при исполнении программного обеспечения комплекса K. *Интеграционное тестирование* использует модель "белого ящика" на модульном уровне. Поскольку тестировщику текст программы известен с детальностью до вызова всех модулей, входящих в тестируемый комплекс, применение структурных критериев на данном этапе возможно и оправдано.

*Интеграционное тестирование* применяется на этапе сборки модульно оттестированных модулей в единый комплекс. Известны два метода *сборки модулей*:

* **Монолитный**, характеризующийся одновременным объединением всех модулей в тестируемый комплекс
* **Инкрементальный**, характеризующийся пошаговым (помодульным) наращиванием комплекса программ с **пошаговым тестированием** собираемого комплекса. В инкрементальном методе выделяют две стратегии добавления модулей:
  + "Сверху вниз" и соответствующее ему *нисходящее тестирование*.
  + "Снизу вверх" и соответственно *восходящее тестирование*.

**Особенности монолитного тестирования** заключаются в следующем: для замены неразработанных к моменту тестирования модулей, кроме самого верхнего ( К на [Рис. 5.1](https://www.intuit.ru/studies/courses/48/48/lecture/1432?page=2#image.5.1)), необходимо дополнительно разрабатывать **драйверы** ( **test driver** ) и/или **заглушки** ( **stub** ), замещающие отсутствующие на момент сеанса тестирования модули нижних уровней.

Сравнение *монолитного* и инкрементального подхода дает следующее:

* *Монолитное тестирование* требует больших трудозатрат, связанных с дополнительной разработкой драйверов и заглушек и со сложностью идентификации ошибок, проявляющихся в пространстве собранного кода.
* Пошаговое тестирование связано с меньшей трудоемкостью идентификации ошибок за счет постепенного наращивания объема тестируемого кода и соответственно локализации добавленной области тестируемого кода.
* *Монолитное тестирование* предоставляет большие возможности распараллеливания работ особенно на начальной *фазе тестирования*.

Особенности *нисходящего тестирования* заключаются в следующем: организация среды для исполняемой очередности вызовов оттестированными модулями тестируемых модулей, постоянная разработка и использование заглушек, организация приоритетного тестирования модулей, содержащих *операции* обмена с окружением, или модулей, критичных для тестируемого алгоритма.

Например, порядок тестирования комплекса K ([Рис. 5.1](https://www.intuit.ru/studies/courses/48/48/lecture/1432?page=2#image.5.1)) при *нисходящем тестировании* может быть таким, как показано в [примере 5.3](https://www.intuit.ru/studies/courses/48/48/lecture/1432?page=2#example.5.3), где тестовый набор, разработанный для модуля Mi, обозначен как XYi = (X, Y)i

Пример 5.3. Возможный порядок тестов при нисходящем тестировании

1) K->XYK

2) M1->XY1

3) M11->XY11

4) M2->XY2

5) M22->XY22

6) M21->XY21

7) M12->XY12

Недостатки *нисходящего тестирования*:

* Проблема разработки достаточно "интеллектуальных" заглушек, т.е. заглушек, пригодных к использованию при моделировании различных режимов работы комплекса, необходимых для тестирования
* Сложность организации и разработки среды для реализации исполнения модулей в нужной последовательности
* Параллельная разработка модулей верхних и нижних уровней приводит к не всегда эффективной реализации модулей из-за подстройки (специализации) еще не тестированных модулей нижних уровней к уже оттестированным модулям верхних уровней

**Особенности восходящего тестирования** в организации порядка *сборки* и перехода к тестированию модулей, соответствующему порядку их реализации.

Например, порядок тестирования комплекса K ([Рис. 5.1](https://www.intuit.ru/studies/courses/48/48/lecture/1432?page=2#image.5.1)) при *восходящем тестировании* может быть следующим ([пример. 5.4](https://www.intuit.ru/studies/courses/48/48/lecture/1432?page=2#example.5.4)).

Пример 5.4. Возможный порядок тестов при восходящем тестировании

1) M11->XY11

2) M12->XY12

3) M1->XY1

4) M21->XY21

5) M2(M21, Stub(M22))->XY2

6) K(M1, M2(M21, Stub(M22)) ->XYK

7) M22->XY22

8) M2->XY2

9) K->XYK

Недостатки *восходящего тестирования*:

* Запаздывание проверки концептуальных особенностей тестируемого комплекса
* Необходимость в разработке и использовании драйверов

#### Особенности интеграционного тестирования для процедурного программирования

Процесс построения набора тестов при структурном тестировании определяется принципом, на котором основывается конструирование Графа *Модели Программы* (ГМП). От этого зависит множество тестовых путей и генерация тестов, соответствующих тестовым путям.

Первым подходом к разработке программного обеспечения является *процедурное (модульное) программирование*. Традиционное *процедурное программирование* предполагает написание исходного кода в императивном (повелительном) стиле, предписывающем определенную последовательность выполнения команд, а также описание программного проекта с помощью функциональной декомпозиции. Такие языки, как Pascal и C, являются императивными. В них порядок исходных строк кода определяет порядок передачи управления, включая последовательное исполнение, выбор условий и повторное исполнение участков программы. Каждый модуль имеет несколько точек входа (при строгом написании кода - одну) и несколько точек выхода (при строгом написании кода - одну). Сложные программные проекты имеют модульно-иерархическое построение, и тестирование модулей является начальным шагом процесса тестирования ПО. Построение *графовой модели* модуля является тривиальной задачей, а тестирование практически всегда проводится по критерию покрытия ветвей C1, т.е. каждая дуга и каждая вершина графа модуля должны содержаться, по крайней мере, в одном из путей тестирования.

Таким образом, M(P,C1) = E Nij, где Е - множество дуг, а Nij - *входные вершины* ГМП.

Сложность тестирования модуля по критерию С1 выражается уточненной формулой для оценки топологической сложности МакКейба:

V(P,C1) = q + kin, где q - число бинарных выборов для условий ветвления, а kin - число входов графа.

Для *интеграционного тестирования* наиболее существенным является рассмотрение *модели программы*, построенной с использованием диаграмм *потоков управления*. Контролируются также связи через данные, подготавливаемые и используемые другими группами программ при взаимодействии с тестируемой группой. Каждая переменная межмодульного интерфейса проверяется на тождественность описаний во взаимодействующих модулях, а также на соответствие исходным программным спецификациям. Состав и структура информационных связей реализованной группы модулей проверяются на соответствие спецификации требований этой группы. Все реализованные связи должны быть установлены, упорядочены и обобщены.

При *сборке модулей* в единый программный комплекс появляется два варианта построения *графовой модели* проекта:

* Плоская или иерархическая модель проекта (например, [Рис. 4.2](https://www.intuit.ru/studies/courses/48/48/lecture/1432?page=2), [Рис. 4.3](https://www.intuit.ru/studies/courses/48/48/lecture/1432?page=2)).
* *Граф вызовов*.

Если программа P состоит из p модулей, то при интеграции модулей в комплекс фактически получается громоздкая плоская ([Рис. 4.2](https://www.intuit.ru/studies/courses/48/48/lecture/1432?page=2)) или более простая - иерархическая ([Рис. 4.3](https://www.intuit.ru/studies/courses/48/48/lecture/1432?page=2)) - модель программного проекта. В качестве критерия тестирования на интеграционном уровне обычно используется критерий покрытия ветвей C1. Введем также следующие обозначения:

|  |
| --- |
| n - число узлов в графе; |
| e - число дуг в графе; |
| q - число бинарных выборов из *условий ветвления* в графе; |
| kin - число входов в граф; |
| kout - число выходов из графов; |
| kext - число точек входа, которые могут быть вызваны извне. |

Тогда сложность *интеграционного тестирования* всей программы P по критерию C1 может быть выражена формулой:

V(P,C1) = \Sigma V(Mod_{i}, C1) - k_{in} +k_{ext} = 
\\
        e - n - k_{ext} + k_{out} = 
\\
        q + k_{ext}, (\forall Mod_{i}\in  P)

Однако при подобном подходе к построению ГМП разработчик тестового набора неизбежно сталкивается с неприемлемо высокой сложностью тестирования V(P,C) для проектов среднего и большого объема (размером в 105 - 107 строк), что следует из роста топологической сложности *управляющего графа* по МакКейбу. Таким образом, используя плоскую или иерархическую модель, трудно дать оценку тестированности TV(P,C,T) для всего проекта и оценку зависимости тестированности проекта от тестированности отдельного модуля TV(Modi,C), включенного в этот проект.

Рассмотрим вторую модель *сборки модулей* в *процедурном программировании* - *граф вызовов*. В этой модели в случае *интеграционного тестирования* учитываются только вызовы модулей в программе. Поэтому из множества M(Modi,C) тестируемых элементов можно исключить те элементы, которые не подвержены влиянию интеграции, т. е. узлы и дуги, не соединенные с вызовами модулей: M(Mod_{i},C') = E' \cup N_{in}, где E' = \{ (n_{i}, n_{j})\in  E | n_{i} или nj содержит вызовы модулей}, т.е. E' - подмножество ребер графа модуля, а Nin - "входные" узлы графа. Эта модификация ГМП приводит к получению нового графа - *графа вызовов*, каждый узел в этом графе представляет модуль (процедуру), а каждая дуга - вызов модуля (процедуры). Для *процедурного программирования* подобный шаг упрощает графовую модель программного проекта до приемлемого уровня сложности. Таким образом, может быть определена цикломатическая сложность упрощенного графа модуля Modi как V'(Modi,C'), а громоздкая формула, выражающая сложность *интеграционного тестирования* программного проекта, принимает следующий вид:

V'(P,C1') = \Sigma  V'(Mod_{i}, C1') - k_{in} +k_{ext}

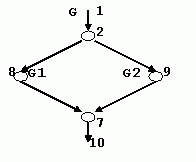
Так, для программы, ГМП которой приведена на [Рис. 4.2](https://www.intuit.ru/studies/courses/48/48/lecture/1432?page=2), для получения *графа вызовов* из иерархической модели проекта должны быть исключены все дуги, кроме:

1. Дуги 1-2, содержащей входной узел 1 графа G.
2. Дуг 2-8, 8-7, 7-10, содержащих вызов модуля G1.
3. Дуг 2-9, 9-7, 7-10, содержащих вызов модуля G2.

В результате *граф вызовов* примет вид, показанный на [Рис. 5.2](https://www.intuit.ru/studies/courses/48/48/lecture/1432?page=2#image.5.2), а сложность данного графа по критерию C1' равна:

V'(G,C1') = q + Kext =1+1=2.

V'(Modi,C') также называется в литературе сложностью модульного дизайна (complexity of module design).

**Рис. 5.2.**Граф вызовов модулей

Сумма сложностей модульного дизайна для всех модулей по критерию С1 или сумма их аналогов для других критериев тестирования, исключая значения модулей самого нижнего уровня, дает сложность *интеграционного тестирования* для *процедурного программирования*.

### Особенности интеграционного тестирования для объектно-ориентированного программирования

Программный проект, написанный в соответствии с объектно-ориентированным подходом, будет иметь ГМП, существенно отличающийся от ГМП традиционной "процедурной" программы. Сама разработка проекта строится по другому принципу - от определения классов, используемых в программе, построения дерева классов к реализации кода проекта. При правильном использовании классов, точно отражающих прикладную область приложения, этот метод дает более короткие, понятные и легко контролируемые программы.

Объектно-ориентированное *программное обеспечение* является событийно управляемым. Передача управления внутри программы осуществляется не только путем явного указания последовательности обращений одних функций программы к другим, но и путем генерации сообщений различным объектам, разбора сообщений соответствующим обработчиком и передача их объектам, для которых данные сообщения предназначены. Рассмотренная ГМП в данном случае становится неприменимой. Эта модель, как *минимум*, требует адаптации к требованиям, вводимым объектно-ориентированным подходом к написанию программного обеспечения. При этом происходит переход от модели, описывающей структуру программы, к модели, описывающей поведение программы, что для тестирования можно классифицировать как положительное свойство данного перехода. Отрицательным аспектом совершаемого перехода для применения рассмотренных ранее моделей является потеря заданных в явном виде связей между модулями программы.

Перед тем как приступить к описанию *графовой модели* объектно-ориентированной программы, остановимся отдельно на одном существенном аспекте разработки программного обеспечения на языке объектно-ориентированного программирования (*ООП*), например, C++ или С#. *Разработка программного обеспечения* высокого качества для MS *Windows* или любой другой операционной системы, использующей стандарт "*look* *and* *feel*", с применением только вновь созданных классов практически невозможна. Программист должен будет затратить массу времени на решение стандартных задач по созданию пользовательского интерфейса. Чтобы избежать работы над давно решенными вопросами, во всех современных компиляторах предусмотрены специальные библиотеки классов. Такие библиотеки включают в себя практически весь программный *интерфейс* операционной системы и позволяют задействовать при программировании средства более высокого уровня, чем просто вызовы функций. Базовые конструкции и классы могут быть переиспользованы при разработке нового программного проекта. За счет этого значительно сокращается время разработки приложений. В качестве примера подобной системы можно привести библиотеку Microsoft *Foundation* *Class* для компилятора MS *Visual* C++.

Работа по тестированию приложения не должна включать в себя проверку работоспособности элементов библиотек, ставших фактически промышленным стандартом для разработки программного обеспечения, а только проверку кода, написанного непосредственно разработчиком программного проекта. Тестирование объектно-ориентированной программы должно включать те же уровни, что и тестирование процедурной программы - модульное, интеграционное и системное. Внутри класса отдельно взятые методы имеют *императивный* характер исполнения. Все языки *ООП* возвращают *контроль* вызывающему объекту, когда сообщение обработано. Поэтому каждый метод (*функция* - член класса) должен пройти традиционное *модульное тестирование* по выбранному критерию C (как правило, С1 ). В соответствии с введенными выше обозначениями, назовем метод Modi, а сложность тестирования - V(Modi,C). Все результаты, полученные в лекции 5 для тестирования модулей, безусловно, подходят для тестирования методов классов. Каждый *класс* должен быть рассмотрен и как субъект *интеграционного тестирования*. *Интеграция* для всех методов класса проводится с использованием инкрементальной стратегии снизу вверх. При этом мы можем переиспользовать тесты для классов-родителей тестируемого класса, что следует из принципа наследования - от базовых классов, не имеющих родителей, к самым верхним уровням классов.

Графовая модель класса, как и объектно-ориентированной программы, на интеграционном уровне в качестве узлов использует методы. Дуги данной ГМП (вызовы методов) могут быть образованы двумя способами:

* Прямым вызовом одного метода из кода другого, в случае, если вызываемый метод виден (не закрыт для доступа средствами языка программирования) из класса, содержащего вызывающий метод, присвоим такой конструкции название ***Р-путь (P-path, Procedure path, процедурный путь)*** .
* Обработкой сообщения, когда явного вызова метода нет, но в результате работы "вызывающего" метода порождается сообщение, которое должно быть обработано "вызываемым" методом.

Для второго случая "вызываемый" метод может породить другое сообщение, что приводит к возникновению цепочки исполнения последовательности методов, связанных сообщениями. Подобная цепочка носит название *ММ-путь (MM-path, Metod/Message path, путь метод/сообщение)*. *ММ-путь* заканчивается, когда достигается метод, который при отработке не вырабатывает новых сообщений (т. е. вырабатывает "сообщение покоя").

Пример *ММ-путей* приведен на [рисунке 6.1](https://www.intuit.ru/studies/courses/48/48/lecture/1434?page=1#image.6.1). Данная конструкция отражает событийно управляемую природу объектно-ориентированного программирования и может быть взята в качестве основы для построения *графовой модели* класса или объектно-ориентированной программы в целом. На [рисунке 6.1](https://www.intuit.ru/studies/courses/48/48/lecture/1434?page=1#image.6.1) можно выделить четыре *ММ-пути* (1-4) и один *P-путь* (5):

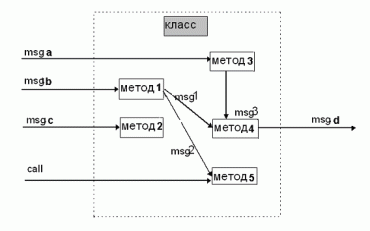
1. msg a \to метод 3 \to msg 3 \to метод 4 \to msg d
2. msg b \to метод 1 \to msg 1 \to метод 4 \to msg d
3. msg b \to метод 1 \to msg 2 \to метод 5
4. msg c \to метод 2
5. call \to метод 5

Здесь *класс* изображен как объединенное множество методов.

Введем следующие обозначения:

Kmsg - число методов класса, обрабатывающих различные сообщения;

Kem - число методов класса, которые не закрыты от прямого вызова из других классов программы.

**Рис. 6.1.**Пример MM-путей и P-путей в графовой модели класса

Если рассматривать *класс* как программу P, то можно выделить следующие отличия от программы, построенной по процедурному принципу:

* Значение Kext (число точек входа, которые могут быть вызваны извне) определяется как сумма методов - обработчиков сообщений Kmsg (например, в MS Visual C++ обозначаются зарезервированным словом afx\_msg и используются для работы с картой сообщений класса) и тех методов, которые могут быть вызваны из других классов программы Kem. Это определяется самим разработчиком путем разграничения доступа к методам класса (с помощью ключевых слов разграничения доступа public, private, protected ) при написании методов, а также назначении дружественных ( friend ) функций и дружественных классов. Таким образом, Kext = Kmsg + Kem, и имеет новый по сравнению с процедурным программированием физический смысл.
* Принцип соединения узлов в ГМП, отражающий два возможных типа вызовов методов класса (через *ММ-пути* и *Р-пути* ), что приводит к новому наполнению для множества М требуемых элементов.
* Методы (модули) непрозрачны для внешних объектов, что влечет за собой неприменимость механизма упрощения графа модуля, используемого для получения *графа вызовов* в *процедурном программировании*.

С учетом приведенных замечаний, информационные связи между модулями программного проекта получают новый физический смысл, а формула *оценки сложности интеграционного тестирования* класса Cls принимает вид: V(Cls, C) = f (Kmsg, Kem)

В ходе *интеграционного тестирования* должны быть проверены все возможные внешние вызовы методов класса, как непосредственные обращения, так и вызовы, инициированные получением сообщений

*Значение* числа *ММ-путей* зависит от схемы обработки сообщений данным классом, что должно быть определено в спецификации класса. Например, для класса, изображенного в [примере 5.4](https://www.intuit.ru/studies/courses/48/48/lecture/1434?page=2), сложность *интеграционного тестирования* V(Cls,C)=5 (множество неизбыточных тестов Т для класса составляют 4 *ММ-пути* плюс внешний вызов метода 5, т. е. *Р-путь* ).

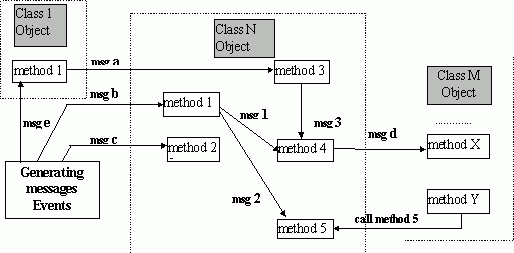
Данные - члены класса (данные, описанные в самом классе, и унаследованные от классов-родителей видимые извне данные) рассматриваются как "*глобальные переменные*", они должны быть протестированы отдельно на основе принципов тестирования потоков данных.

Когда *класс* программы P протестирован, *объект* данного класса может быть включен в общий *граф* G программного проекта, содержащий все *ММ-пути* и все вызовы методов классов и процедур, возможные в программе [рис. 6.2](https://www.intuit.ru/studies/courses/48/48/lecture/1434?page=2#image.6.2)

*Программа* P, содержащая n классов, имеет сложность *интеграционного тестирования* классов

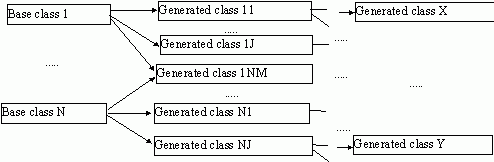
V(P, C) =\varSigma V(Cls_{i}, C)

Формальным представлением описанного выше подхода к тестированию программного проекта служит ***классовая модель программного проекта***, состоящая из дерева классов проекта [рис. 6.3](https://www.intuit.ru/studies/courses/48/48/lecture/1434?page=2#image.6.3) и модели каждого класса, входящего в программный проект [рис. 6.4](https://www.intuit.ru/studies/courses/48/48/lecture/1434?page=2#image.6.4).

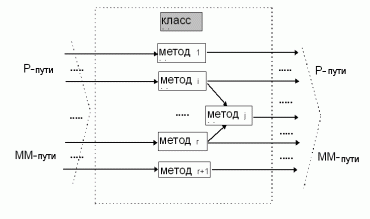


**Рис. 6.2.**Пример включения объекта в модель программного проекта, построенного с использованием MM-путей и P-путей

Таким образом и определяется *классовая модель* проекта для тестирования объектно-ориентированной программы. Как будет показано в дальнейшем, она поддерживает итерационный инкрементальный процесс разработки программного обеспечения.



**Рис. 6.3.**Дерево классов проекта



**Рис. 6.4.**Модель класса, входящего в программный проект

*Методика проведения тестирования программы*, представленной в виде *классовой модели* программного проекта, включает в себя несколько этапов, соответствующих уровням тестирования [рис. 6.5](https://www.intuit.ru/studies/courses/48/48/lecture/1434?page=2#image.6.5):

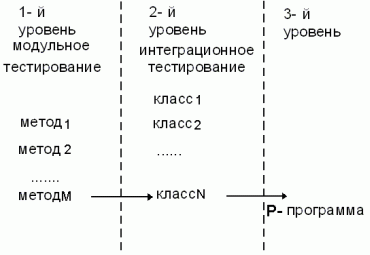
1. На первом уровне проводится тестирование методов каждого класса программы, что соответствует этапу модульного тестирования.
2. На втором уровне тестируются методы класса, которые образуют контекст *интеграционного тестирования* каждого класса.
3. На третьем уровне протестированный класс включается в общий контекст (дерево классов) программного проекта. Здесь становится возможным отслеживать реакцию программы на внешние события

Второй и третий уровни рассматриваемой модели соответствуют этапу *интеграционного тестирования*.

Для третьего уровня важным оказывается понятие атомарной системной функции (АСФ). АСФ - это множество, состоящее из внешнего события на входе системы, реакции системы на это событие в виде одного или более *ММ-путей* и внешнего события на выходе системы. В общем случае внешнее выходное событие может быть нулевым, т. е. неаккуратно написанное *программное обеспечение* может не обеспечивать внешней реакции на действия пользователя. АСФ, состоящая из входного внешнего события, одного *ММ-пути* и выходного внешнего события, может быть взята в качестве модели для нити (*thread*). Тестирование подобной АСФ в рамках *классовой модели* ГМП реализуется довольно сложно, так как хотя динамическое взаимодействие нитей (потоков) в процессе исполнения естественно фиксируется в log-файлах, запоминающих результаты трассировки исполнения программ, оно же достаточно сложно отображается на классовой ГМП. Причина в том, что *классовая модель* ориентирована на *отображение* статических характеристик проекта, а в данном случае требуется *отображение* поведенческих характеристик. Как правило, тестирование взаимодействия нитей в ходе исполнения программного комплекса выносится на уровень *системного тестирования* и использует другие более приспособленные для описания поведения модели. Например, описание поведения программного комплекса средствами *языков спецификаций* *MSC*, SDL, *UML*.

Явный учет границ между интеграционным и системным уровнями тестирования дает преимущество при планировании *работ* на *фазе тестирования*, а возможность сочетать различные методы и критерии тестирования в ходе работы над программным проектом дает наилучшие результаты.

*Объектно-ориентированный подход*, ставший в настоящее время неявным стандартом разработки программных комплексов, позволяет широко использовать *иерархическую модель* программного проекта, приведенная на [рис. 6.5](https://www.intuit.ru/studies/courses/48/48/lecture/1434?page=2#image.6.5) схема иллюстрирует способ применения. Каждый *класс* рассматривается как *объект* модульного и *интеграционного тестирования*. Сначала каждый *метод класса* тестируется как *модуль* по выбранному критерию C. Затем *класс* становится объектом *интеграционного тестирования*. Далее осуществляется *интеграция* всех методов всех классов в единую структуру - *классовую модель* проекта, где в общую ГМП протестированные модули входят в виде узлов (интерфейсов вызова) без учета их внутренней структуры, а их детальные описания образуют *контекст* всего программного проекта.



**Рис. 6.5.**Уровни тестирования классовой модели программного проекта

Сама технология объектно-ориентированного программирования (одним из определяющих принципов которой является *инкапсуляция* с возможностью ограничения доступа к данным и методам - членам класса) позволяет применить подобную трактовку вхождения модулей в общую ГМП. При этом тесты для отдельно рассмотренных классов переиспользуются, входя в общий набор тестов для программы P.

### **Системное тестирование**

*Системное тестирование* качественно отличается от интеграционного и модульного уровней. *Системное тестирование* рассматривает тестируемую систему в целом и оперирует на уровне пользовательских интерфейсов, в отличие от последних фаз *интеграционного тестирования*, которое оперирует на уровне интерфейсов модулей. Различны и цели этих уровней тестирования. На уровне системы часто сложно и малоэффективно анализировать прохождение тестовых траекторий внутри программы или отслеживать правильность работы конкретных функций. Основная *задача системного тестирования* - в выявлении дефектов, связанных с работой системы в целом, таких как неверное использование ресурсов системы, непредусмотренные комбинации данных пользовательского уровня, несовместимость с окружением, непредусмотренные сценарии использования, отсутствующая или неверная функциональность, неудобство в применении и тому подобное.

*Системное тестирование* производится над проектом в целом с помощью метода "черного ящика". Структура программы не имеет никакого значения, для проверки доступны только входы и выходы, видимые пользователю. Тестированию подлежат коды и пользовательская документация.

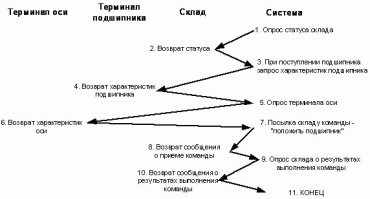
*Категории тестов* *системного тестирования*:

1. Полнота решения функциональных задач.
2. *Стрессовое тестирование* - на предельных объемах нагрузки входного потока.
3. Корректность использования ресурсов (утечка памяти, возврат ресурсов).
4. Оценка производительности.
5. Эффективность защиты от искажения данных и некорректных действий.
6. Проверка инсталляции и конфигурации на разных платформах.
7. Корректность документации

Поскольку *системное тестирование* проводится на пользовательских интерфейсах, создается иллюзия того, что построение специальной системы *автоматизации тестирования* не всегда необходимо. Однако объемы данных на этом уровне таковы, что обычно более эффективным подходом является полная или частичная *автоматизация тестирования*, что приводит к созданию тестовой системы гораздо более сложной, чем *система тестирования*, применяемая на уровне тестирования модулей или их комбинаций.

#### Пример системного тестирования приложения "Поступление подшипника на склад"

В спецификации тестового случая задано состояние окружения (входные данные) и ожидаемая последовательность событий в системе (ожидаемый результат). После прогона тестового случая мы получаем реальную последовательность событий в системе ([пример 7.1](https://www.intuit.ru/studies/courses/48/48/lecture/1436?page=1#example.7.1),[пример 7.3](https://www.intuit.ru/studies/courses/48/48/lecture/1436?page=1#example.7.3)) при заданном состоянии окружения. Сравнивая фактический результат с ожидаемым, можно сделать вывод о том, прошла или не прошла тестируемая система испытание на заданном тестовом случае. В качестве ожидаемого результата будем использовать спецификацию тестового случая, поскольку она определяет, как, для заданного состояния окружения, система должна функционировать.



**Рис. 4-15.**Краткое описание тестируемой системы 'Поступление подшипника на склад'

Спецификация тестового случая №1:

Состояние окружения (входные данные - X ):

Статус склада - 32. Пришел подшипник.

Статус обмена с терминалом подшипника (0 - есть подшипник) и его параметры - "Статус=0 Диаметр=12".

Статус обмена с терминалом оси (1 - нет оси) и ее параметры - "Статус=1 Диаметр=12".

"Статус=1 Диаметр=12".

Статус команды - 0. Команда успешно принята.

Сообщение от склада - 1. Команда успешно выполнена.

Ожидаемая последовательность событий (выходные данные – Y):

Система запрашивает статус склада (вызов функции GetStoreStat ) и получает 32

Система запрашивает параметры подшипника (вызов функции GetRollerPar ) и получает Статус = 0 Диаметр=12

Система запрашивает параметры оси (вызов функции GetAxlePar ) и получает Статус = 1 Диаметр=0

Система добавляет в очередь команд склада на последнее место команду SendR (получить из приемника в ячейку) (вызов функции SendStoreCom ) и получает сообщение о том, что команда успешно принята – статус=0

Система запрашивает склад о результатах выполнения команды (вызов функции GetStoreMessage ) и получает сообщение о том, что команда успешно выполнена - статус = 1

### **Регрессионное тестирование**

*Регрессионное тестирование* - цикл тестирования, который производится при внесении изменений на фазе *системного тестирования* или сопровождения продукта. Главная проблема *регрессионного тестирования* - выбор между полным и частичным *перетестированием* и пополнением тестовых наборов. При частичном *перетестировании* контролируются только те части проекта, которые связаны с измененными компонентами. На ГМП это пути, содержащие измененные узлы, и, как правило, это методы и классы, лежащие выше модифицированных по уровню, но содержащие их в своем контексте

Пропуск огромного объема тестов, характерного для этапа *системного тестирования*, удается осуществить без потери качественных показателей продукта только с помощью регрессионного подхода.

#### Пример регрессионного тестирования

Получив отчет об ошибке, программист анализирует исходный код, находит ошибку, исправляет ее и модульно или интеграционно тестирует результат.

В свою очередь тестировщик, проверяя внесенные программистом изменения, должен:

* Проверить и утвердить исправление ошибки. Для этого необходимо выполнить указанный в отчете тест, с помощью которого была найдена ошибка.
* Попробовать воспроизвести ошибку каким-нибудь другим способом.
* Протестировать последствия исправлений. Возможно, что внесенные исправления привнесли ошибку (наведенную ошибку) в код, который до этого исправно работал.

### **Комбинирование уровней тестирования**

В каждом конкретном проекте должны быть определены задачи, ресурсы и технологии для каждого уровня тестирования таким образом, чтобы каждый из типов дефектов, ожидаемых в системе, был "адресован", то есть в общем наборе тестов должны иметься тесты, направленные на выявление дефектов подобного типа. [Табл. 4.3](https://www.intuit.ru/studies/courses/48/48/lecture/1436?page=3#table.4.3) суммирует характеристики свойств модульного, интеграционного и системного уровней тестирования. Задача, которая стоит перед тестировщиками и менеджерами, заключается в оптимальном распределении ресурсов между всеми тремя типами тестирования. Например, перенесение усилий на *поиск* фиксированного типа дефектов из области системного в область *модульного тестирования* может существенно снизить сложность и *стоимость* всего процесса тестирования.

|  |  |  |  |
| --- | --- | --- | --- |
| Таблица 4.3. Характеристики модульного, интеграционного и *системного тестирования* | | | |
|  | **Модульное** | **Интеграционное** | **Системное** |
| **Типы дефектов** | Локальные дефекты, такие как опечатки в реализации алгоритма, неверные операции, логические и математические выражения, циклы, ошибки в использовании локальных ресурсов, рекурсия и т.п. | Интерфейсные дефекты, такие как неверная трактовка параметров и их формат, неверное использование системных ресурсов и средств коммуникации, и т.п. | Отсутствующая или некорректная функциональность, неудобство использования, непредусмотренные данные и их комбинации, непредусмотренные или неподдерживаемые сценарии работы, ошибки совместимости, ошибки пользовательской документации, ошибки переносимости продукта на различные платформы, проблемы производительности, инсталляции и т.п. |
| **Необходимость в системе тестирования** | Да | Да | Нет (\*) |
| **Цена разработки системы тестирования** | Низкая | Низкая до умеренной | Умеренная до высокой или неприемлемой |
| **Цена процесса тестирования, то есть разработки, прогона и анализа тестов** | Низкая | Низкая | Высокая |

(\*) *прямой* необходимости в системе тестирования нет, но цена процесса *системного тестирования* часто настолько высока, что требует использования систем автоматизации, несмотря на возможно высокую их *стоимость*.