Лекция 6. Модульное тестирование

Задачи и цели модульного тестирования

Каждая сложная программная система состоит из отдельных частей – модулей, выполняющих ту или иную функцию в составе системы. Для того, чтобы удостовериться в корректной работе системы в целом, необходимо вначале протестировать каждый модуль системы в отдельности. В случае возникновения проблем это позволит проще выявить модули, вызвавшие проблему, и устранить соответствующие дефекты в них. Такое тестирование модулей по отдельности получило называние модульного тестирования (unit testing).

Для каждого модуля, подвергаемого тестированию, разрабатывается тестовое окружение, включающее в себя драйвер и заглушки, готовятся тест-требования и тест-планы, описывающие конкретные тестовые примеры.

Основная цель модульного тестирования – удостовериться в соответствии требованиям каждого отдельного модуля системы перед тем, как будет произведена его интеграция в состав системы.

При этом в ходе модульного тестирования решаются четыре основные задачи.

1.Поиск и документирование несоответствий требованиям – это классическая задача тестирования, включающая в себя не только разработку тестового окружения и тестовых примеров, но и выполнение тестов, протоколирование результатов выполнения, составление отчетов о проблемах.

2.Поддержка разработки и рефакторинга низкоуровневой архитектуры системы и межмодульного взаимодействия – эта задача больше свойственна "легким" методологиям типа XP, где применяется принцип тестирования перед разработкой (Test-driven development), при котором основным источником требований для программного модуля является тест, написанный до самого модуля. Однако, даже при классической схеме тестирования модульные тесты могут выявить проблемы в дизайне системы и нелогичные или запутанные механизмы работы с модулем.

3.Поддержка рефакторинга модулей – эта задача связана с поддержкой процесса изменения системы. Достаточно часто в ходе разработки требуется проводить рефакторинг модулей или их групп – оптимизацию или полную переделку программного кода с целью повышения его сопровождаемости, скорости работы или надежности. Модульные тесты при этом являются мощным инструментом для проверки того, что новый вариант программного кода работает в точности так же, как и старый.

4.Поддержка устранения дефектов и отладки — эта задача сопряжена с обратной связью, которую получают разработчики от тестировщиков в виде отчетов о проблемах. Подробные отчеты о проблемах, составленные на этапе модульного тестирования, позволяют локализовать и устранить многие дефекты в программной системе на ранних стадиях ее разработки или разработки ее новой функциональности.

В силу того, что модули, подвергаемые тестированию, обычно невелики по размеру, модульное тестирование считается наиболее простым (хотя и достаточно трудоемким) этапом тестирования системы. Однако, несмотря на внешнюю простоту, с модульным тестированием связано две проблемы.

1.Не существует единых принципов определения того, что в точности является отдельным модулем.

2.Различия в трактовке самого понятия модульного тестирования – понимается ли под ним обособленное тестирование модуля, работа которого поддерживается только тестовым окружением, или речь идет о проверке корректности работы модуля в составе уже разработанной системы. В последнее время термин "модульное тестирование" чаще используется во втором смысле, хотя в этом случае речь скорее идет об интеграционном тестировании.

Эти две проблемы рассмотрены в двух следующих разделах.

9.2.1.2. Понятие модуля и его границ. Тестирование классов

Традиционное определение модуля с точки зрения его тестирования: "модуль – это компонент минимального размера, который может быть независимо протестирован в ходе верификации программной системы". В реальности часто возникают проблемы с тем, что считать модулем. Существует несколько подходов к данному вопросу:

•модуль – это часть программного кода, выполняющая одну функцию с точки зрения функциональных требований;

•модуль – это программный модуль, т.е. минимальный компилируемый элемент программной системы;

•модуль – это задача в списке задач проекта (с точки зрения его менеджера);

•модуль – это участок кода, который может уместиться на одном экране или одном листе бумаги;

•модуль – это один класс или их множество с единым интерфейсом.

Обычно за тестируемый модуль принимается либо программный модуль (единица компиляции) в случае, если система разрабатывается на процедурном языке программирования, либо класс, если система разрабатывается на объектно-ориентированном языке.

В случае систем, написанных на процедурных языках, процесс тестирования модуля происходит так, как это было рассмотрено в темах 2-4 – для каждого модуля разрабатывается тестовый драйвер, вызывающий функции модуля и собирающий результаты их работы, и набор заглушек, которые имитируют поведение функций, содержащихся в других модулях и не попадающих под тестирование данного модуля. При тестировании объектно-ориентированных систем существует ряд особенностей, прежде всего вызванных инкапсуляцией данных и методов в классах.

В случае объектно-ориентированных систем более мелкое деление классов и использование отдельных методов в качестве тестируемых модулей нецелесообразно в связи с тем, что для тестирования каждого метода потребуется разработка тестового окружения, сравнимого по сложности с уже написанным программным кодом класса. Кроме того, декомпозиция класса нарушает принцип инкапсуляции, согласно которому объекты каждого класса должны вести себя как единое целое с точки зрения других объектов.

Процесс тестирования классов как модулей иногда называют компонентным тестированием. В ходе такого тестирование проверяется взаимодействие методов внутри класса и правильность доступа методов к внутренним данным класса. При таком тестировании возможно обнаружение не только стандартных дефектов, связанных с выходами за границы диапазона или неверно реализованными требованиями, а также обнаружение специфических дефектов объектно-ориентированного программного обеспечения:

•дефектов инкапсуляции, в результате которых, например, сокрытые данные класса оказывается недоступными при помощи соответствующих публичных методов;

•дефектов наследования, при наличии которых схема наследования блокирует важные данные или методы от классов-потомков;

•дефектов полиморфизма, при которых полиморфное поведение класса оказывается распространенным не на все возможные классы;

•дефектов инстанцирования, при которых во вновь создаваемых объектах класса не устанавливаются корректные значения по умолчанию параметров и внутренних данных класса.

Однако, выбор класса в качестве тестируемого модуля имеет и ряд сопряженных проблем.

Определение степени полноты тестирования класса. В том случае, если в качестве тестируемого модуля выбран класс, не совсем ясно, как определять степень полноты его тестирования. С одной стороны, можно использовать классический критерий полноты покрытия программного кода тестами: если полностью выполнены все структурные элементы всех методов, как публичных, так и скрытых, — то тесты можно считать полными.

Однако существует альтернативный подход к тестированию класса, согласно которому все публичные методы должны предоставлять пользователю данного класса согласованную схему работы и достаточно проверить типичные корректные и некорректные сценарии работы с данным классом. Т.е., например, в классе, объекты которого представляют записи в телефонной книжке, одним из типичных сценариев работы будет "Создать запись искать запись и найти ее удалить запись искать запись вторично и получить сообщение об ошибке".

Различия в этих двух методах напоминают различия между тестированием "черного" и "белого" ящиков, но на самом деле второй подход отличается от "черного ящика" тем, что функциональные требования к системе могут быть составлены на уровне более высоком, чем отдельные классы, и установление адекватности тестовых сценариев требованиям остается на откуп тестировщику.

Протоколирование состояний объектов и их изменений. Некоторые методы класса предназначены не для выдачи информации пользователю, а для изменения внутренних данных объекта класса. Значение внутренних данных объекта определяет его состояние в каждый отдельный момент времени, а вызов методов, изменяющих данные, изменяет и состояние объекта. При тестировании классов необходимо проверять, что класс адекватно реагирует на внешние вызовы в любом из состояний. Однако, зачастую из-за инкапсуляции данных невозможно определить внутреннее состояние класса программными способами внутри драйвера.

В этом случае может помочь составление схемы поведения объекта как конечного автомата с определенным набором состояний (подобно тому, как это было описано в теме 2 в разделе "Генераторы сигналов. Событийно-управляемый код"). Такая схема может входить в низкоуровневую проектную документацию (например, в составе описания архитектуры системы), а может составляться тестировщиком или разработчиком на основе функциональных требований к системе. В последнем случае для определения всех возможных состояний может потребоваться ручной анализ программного кода и определение его соответствия требованиям. Автоматизированное тестирование в этом

случае может лишь определить, по всем ли выявленным состояниям осуществлялись переходы и все ли возможные реакции проверялись.

Тестирование изменений. Как уже упоминалось выше, модульные тесты – мощный инструмент проверки корректности изменений, внесенных в исходный код при рефакторинге. Однако, в результате рефакторинга только одного класса, как правило, не меняется его внешний интерфейс с другими классами (интерфейсы меняются при рефакторинге сразу нескольких классов). В результате обычных эволюционных изменений системы у класса может меняться внешний интерфейс, причем как по формальным (изменяются имена и состав методов, их параметры), так и по функциональным признакам (при сохранении внешнего интерфейса меняется логика работы методов). Для проведения модульного тестирования класса после таких изменений потребуется изменение драйвера и, возможно, заглушек. Но только модульного тестирования в данном случае недостаточно, необходимо также проводить и интеграционное тестирование данного класса вместе со всеми классами, которые связаны с ним по данным или по управлению.

Вне зависимости от того, на какие модули, подвергаемые тестированию, разбивается система, рекомендуется изложить принципы выделения тестируемых модулей в плане и стратегии тестирования, а также составить на базе структурной схемы архитектуры системы новую структурную схему, на которой отметить все тестируемые модули. Это позволит спрогнозировать состав и сложность драйверов и заглушек, требуемых для модульного тестирования системы. Такая схема также может использоваться позже на этапе модульного тестирования для выделения укрупненных групп модулей, подвергаемых интеграции.

9.2.1.3. Подходы к проектированию тестового окружения

Вне зависимости от того, какая минимальная единица исходных кодов системы выбирается за минимальный тестируемый модуль, существует еще одно различие в подходах к модульному тестированию.

Первый подход к модульному тестированию основывается на предположении, что функциональность каждого вновь разработанного модуля должна проверяться в автономном режиме без его интеграции с системой. Здесь для каждого вновь разрабатываемого модуля создается тестовый драйвер и заглушки, при помощи которых выполняется набор тестов. Только после устранения всех дефектов в автономном режиме производится интеграция модуля в систему и проводится тестирование на следующем уровне. Достоинством данного подхода является более простая локализация ошибок в модуле, поскольку при автономном тестировании исключается влияние остальных частей системы, которое может вызывать маскировку дефектов (эффект четного числа ошибок). Основной недостаток данного метода – повышенная трудоемкость написания драйверов и заглушек, поскольку заглушки должны адекватно моделировать поведение системы в различных ситуациях, а драйвер должен не только создавать тестовое окружение, но и имитировать внутреннее состояние системы, в составе которой должен функционировать модуль.

Второй подход построен на предположении, что модуль все равно работает в составе системы и если модули интегрировать в систему по одному, то можно протестировать поведение модуля в составе всей системы. Этот подход свойственен большинству современных "облегченных" методологий разработки, в том числе и XP.

В результате применения такого подхода резко сокращаются трудозатраты на разработку заглушек и драйверов – в роли заглушек выступает уже оттестированная часть системы, а драйвер выполняет только функции передачи и приема данных, не моделируя внутреннее состояние системы.

Тем не менее, при использовании данного метода возрастает сложность написания тестовых примеров – для приведения в нужное состояние системы заглушек, как правило, требуется только установить значения тестовых переменных, а для приведения в нужное состояние части реальной системы необходимо выполнить целый сценарий. Каждый тестовый пример в этом случае должен содержать такой сценарий.

Кроме того, при этом подходе не всегда удается локализовать ошибки, скрытые внутри модуля, которые могут проявиться при интеграции следующих модулей.

Замечание. О том, что такое Reflection, можно прочесть на http://msdn2.microsoft.com/en-us/library/cxz4wk15(VS.80).aspx

Модульное тестирование позволяет:

находить баги на самом раннем этапе;

автоматизировать процесс исполнения тест-кейсов;

реализовать процедуру регрессионного тестирования (быстро проверить, не привело ли очередное изменение кода к регрессии);

облегчить документирование кода (модульный тест-кейс – «живой help» для тестируемого модуля);

тестировать базовую функциональность без GUI (например, проводить нагрузочные тесты, не прибегая к инструментам внешнего доступа);

улучшить качество кода, сделать его максимально «чистым» (при использовании в проекте процедуры TDD).

JUnit – библиотека модульного тестирования из семейства xUnit для языка JAVA

Некоторые полезные аннотации JUnit 4

@Test

Обозначает сами тестовые методы (тест-кейсы). Есть два необязательных параметра, expected — задает ожидаемое исключение и timeout — задает время, по истечению которого тест считается провалившимся.

@Before

Обозначает методы, которые будут вызваны до исполнения тестовых методов. Здесь обычно размещаются предустановки для тестов, например - генерация тестовых данных.

@After

Обозначает методы, которые будут вызваны после выполнения тестовых методов. Здесь размещаются операции освобождения ресурсов после тестов, например - очистка тестовых данных.

@Ignore

Применяется для исключения какого-либо метода из процедуры тестирования

@Rule

Позволяет задавать специфические установки тестирования, которые будут выполнены до, после и во время выполнения тестовых методов.

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 на основе потока данных, т. е. информационного графа программы.

Тестирование на основе потока управления. Особенности использования структурных критериев тестирования С0,С1,С2 были рассмотрены в разделе 2. 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 впервые был описан в 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 CP, 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.

Пример модульного тестирования

Предлагается протестировать класс TCommand, который реализует команду для склада. Этот класс содержит единственный метод TCommand.GetFullName(), спецификация которого описана (Практикум, Приложение 2 HLD) следующим образом:

	...
	Операция GetFullName() возвращает полное имя команды, соответствующее ее допустимому коду, указанному в поле NameCommand. В противном случае возвращается сообщение "ОШИБКА : Неверный код команды". Операция может быть применена в любой момент.
	...

Разработаем спецификацию тестового случая для тестирования метода GetFullName на основе приведенной спецификации класса

Таблица 5.1. Спецификация теста

Название класса: TCommand	**Название тестового случая:** TCommandTest1
Описание тестового случая: Тест проверяет правильность работы метода GetFullName - получения полного названия команды на основе кода команды. В тесте подаются следующие значения кодов команд (входные значения): -1, 1, 2, 4, 6, 20, (причем -1 - запрещенное значение).	
Начальные условия: Нет.	
Ожидаемый результат: Перечисленным входным значениям должны соответствовать следующие выходные: Коду команды -1 должно соответствовать сообщение "ОШИБКА: Неверный код команды" Коду команды 1 должно соответствовать полное название команды "ПОЛУЧИТЬ ИЗ ВХОДНОЙ ЯЧЕЙКИ" Коду команды 2 должно соответствовать полное название команды "ОТПРАВИТЬ ИЗ ЯЧЕЙКИ В ВЫХОДНУЮ ЯЧЕЙКУ" Коду команды 4 должно соответствовать полное название команды "ПОЛОЖИТЬ В РЕЗЕРВ" Коду команды 6 должно соответствовать полное название команды "ПРОИЗВЕСТИ ЗАНУЛЕНИЕ" Коду команды 20 должно соответствовать полное название команды "ЗАВЕРШЕНИЕ КОМАНД ВЫДАЧИ"	

Для тестирования метода класса TCommand.GetFullName() был создан тестовый драйвер - класс TCommandTester. Класс TCommandTester содержит метод TCommandTest1(), в котором реализована вся функциональность теста. В данном случае для покрытия спецификации достаточно перебрать следующие значения кодов команд: -1, 1, 2, 4, 6, 20, (-1 - запрещенное значение) и получить соответствующее им полное название команды с помощью метода

GetFullName() (Пары значений (X, Yв) при исполнении теста заносятся в log-файл для последующей проверки на соответствие спецификации.
После завершения теста следует просмотреть журнал теста, чтобы сравнить полученные результаты с ожидаемыми, заданными в спецификации тестового случая TCommandTest1
```
class TCommandTester:Tester // Тестовый драйвер
{
...
TCommand OUT;
public TCommandTester()
{
OUT=new TCommand();
Run();
}
private void Run()
{
TCommandTest1();
}
private void TCommandTest1()
{
int[] commands = {-1, 1, 2, 4, 6, 20};
for(int i=0;i<=5;i++)
{
OUT.NameCommand=commands[i];
LogMessage(commands[i].ToString()+
" : "+OUT.GetFullName());
}
}
...
}
```
Пример 5.1. Тестовый драйвер
```
-1 : ОШИБКА : Неверный код команды
1 : ПОЛУЧИТЬ ИЗ ВХОДНОЙ ЯЧЕЙКИ
2 : ОТПРАВИТЬ ИЗ ЯЧЕЙКИ В ВЫХОДНУЮ ЯЧЕЙКУ
4 : ПОЛОЖИТЬ В РЕЗЕРВ
6 : ПРОИЗВЕСТИ ЗАНУЛЕНИЕ
20 : ЗАВЕРШЕНИЕ КОМАНД ВЫДАЧИ
```
Пример 5.2. Спецификация классов тестовых случаев

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.

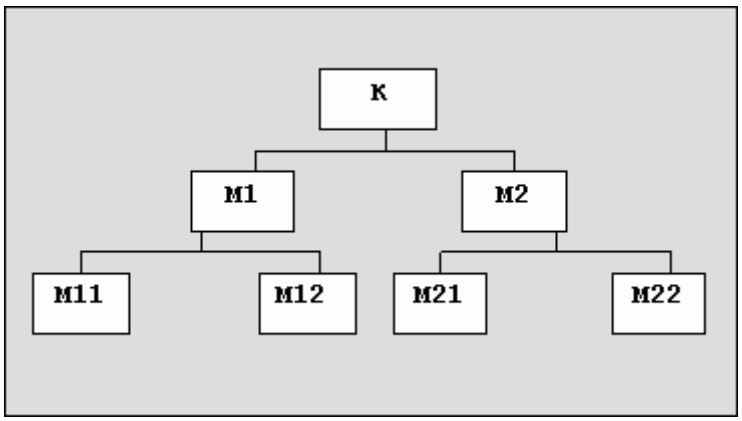

Рис. 5.1. Пример структуры комплекса программ

На Рис. 5.1 приведена структура комплекса программ К, состоящего из оттестированных на этапе модульного тестирования модулей М1, М2, М11, М12, М21, М22. Задача, решаемая методом интеграционного тестирования, - тестирование межмодульных связей, реализующихся при исполнении программного обеспечения комплекса К. 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 восходящее тестирование.
    - "Снизу вверх" и соответственно нисходящее тестирование.

Особенности монолитного тестирования заключаются в следующем: для замены неразработанных к моменту тестирования модулей, кроме самого верхнего (К на Рис. 5.1), необходимо дополнительно разрабатывать **драйверы** (**test driver**) и/или **заглушки** (**stub**) [9], 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 К (Рис. 5.1) при нисходящем тестировании может быть таким, как показано в примере 5.3, где тестовый набор, разработанный для модуля M_i, обозначен как $XY_i = (X, Y)_i$

1) $K \rightarrow XY_K$
2) $M_1 \rightarrow XY_1$
3) $M_{11} \rightarrow XY_{11}$
4) $M_2 \rightarrow XY_2$
5) $M_{22} \rightarrow XY_{22}$
6) $M_{21} \rightarrow XY_{21}$
7) $M_{12} \rightarrow XY_{12}$

Пример 5.3. Возможный порядок тестов при нисходящем тестировании

Недостатки нисходящего тестирования:

- Проблема разработки достаточно "интеллектуальных" заглушек, т.е. заглушек, способных 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) при восходящем тестировании может быть следующим (пример. 5.4).
1) M_{11}->XY_{11}
2) M_{12}->XY_{12}
3) M_1->XY_1
4) M_{21}->XY_{21}
5) $M_2(M_{21}, Stub(M_{22}))$->$XY_2$
6) $K(M_1, M_2(M_{21}, Stub(M_{22}))$ ->XY_K
7) M_{22}->XY_{22}
8) M_2->XY_2
9) K->XY_K

Пример 5.4. Возможный порядок тестов при восходящем тестировании
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, $M(P,C1) = E\ N_{ij}$, где E - множество дуг, а N_{ij} - входные вершины ГМП.
Сложность тестирования модуля по критерию C1 выражается уточненной формулой для оценки топологической сложности МакКейба:
$V(P,C1) = q + k_{in}$, где q - число бинарных выборов для условий ветвления, а k_{in} - 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, Рис. 4.3).
- Граф вызовов.

Если программа P состоит из p модулей, то при интеграции модулей в комплекс фактически получается громоздкая плоская (Рис. 4.2) или более простая - иерархическая (Рис. 4.3) - 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;
k_{in} - число входов в граф;
k_{out} - число выходов из графов;
k_{ext} - число точек входа, которые могут быть вызваны извне.

Тогда сложность интеграционного тестирования всей программы P по критерию C1 может быть выражена формулой [17]:
$V(P,C1) = \Sigma V(Mod_i, C1) - k_{in} + k_{ext} =$
$e - n - k_{ext} + k_{out} =$
$q + k_{ext}, (\forall Mod_i \in P)$
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 10^5 - 10^7 строк) [18], 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 $TV(Mod_i,C)$, 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 $M(Mod_i,C)$ 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 $E' = \{(n_i, n_j) \in E \mid n_i$ или n_j содержит вызовы модулей}, т.е. E' - подмножество ребер графа модуля, а N_{in} - "входные" узлы графа [17]. 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 Mod_i как $V'(Mod_i,C')$, а громоздкая формула, выражающая сложность интеграционного тестирования программного проекта, принимает следующий вид [19]:
$V'(P,C1') = \Sigma V'(Mod_i, C1') - k_{in} + k_{ext}$
Так, для программы, ГМП которой приведена на Рис. 4.2, 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, а сложность данного графа по критерию C1' равна:
$V'(G,C1') = q + K_{ext} = 1+1=2.$
$V'(Mod_i,C')$ также называется в литературе сложностью модульного дизайна (complexity of module design)

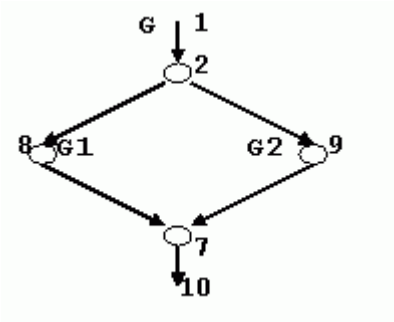

Рис. 5.2. Граф вызовов модулей

Сумма сложностей модульного дизайна для всех модулей по критерию C1 или сумма их аналогов для других критериев тестирования, исключая значения модулей самого нижнего уровня, дает сложность интеграционного тестирования для процедурного программирования.