Лекция 7 Интеграционное тестирование. Нисходящее и восходящее тестирование программ

Тема данной лекции - процесс интеграционного тестирования, его задачи и цели. Рассматриваются организационные аспекты интеграционного тестирования - структурная и временная классификации методов интеграционного тестирования, планирование интеграционного тестирования. Цель данной лекции: дать представление о процессе интеграционного тестирования, его технической и организационной составляющих

Интеграцио́нное тести́рование (англ. *Integration testing*, иногда называется англ. *Integration and Testing*, аббревиатура англ. *I&T*) — одна из фаз тестирования программного обеспечения, при которой отдельные программные модули объединяются и тестируются в группе. Обычно интеграционное тестирование проводится после модульного тестирования и предшествует системному тестированию.

Интеграционное тестирование в качестве входных данных использует модули, над которыми было проведено модульное тестирование, группирует их в более крупные множества, выполняет тесты, определённые в плане тестирования для этих множеств, и представляет их в качестве выходных данных и входных для последующего системного тестирования.

Целью интеграционного тестирования является проверка соответствия проектируемых единиц функциональным, приёмным и требованиям надежности. Тестирование этих проектируемых единиц — объединения, множества или группы модулей — выполняется через их интерфейс, с использованием тестирования «чёрного ящика».

Системы непрерывной интеграции[править | править код]

Для автоматизации интеграционного тестирования применяются ***системы непрерывной интеграции*** (англ. *Continuous Integration System*, CIS). Принцип действия таких систем состоит в следующем:

1. CIS производит мониторинг системы контроля версий;
2. При изменении исходных кодов в репозитории производится обновление локального хранилища;
3. Выполняются необходимые проверки и модульные тесты;
4. Исходные коды компилируются в готовые выполняемые модули;
5. Выполняются тесты интеграционного уровня;
6. Генерируется отчет о тестировании.

Таким образом, автоматические интеграционные тесты выполняются сразу же после внесения изменений, что позволяет обнаруживать и устранять ошибки в короткие сроки.

Результатом тестирования и верификации отдельных модулей, составляющих программную систему, должно быть заключение о том, что эти модули являются внутренне непротиворечивыми и соответствуют требованиям. Однако отдельные модули редко функционируют сами по себе, поэтому следующая задача после тестирования отдельных модулей - тестирование корректности взаимодействия нескольких модулей, объединенных в единое целое. Такое тестирование называют интеграционным. Его цель - удостовериться в корректности совместной работы компонент системы.

Интеграционное тестирование называют еще тестированием архитектуры системы. С одной стороны, это название обусловлено тем, что интеграционные тесты включают в себя проверки всех возможных видов взаимодействий между программными модулями и элементами, которые определяются в архитектуре системы - таким образом, интеграционные тесты проверяют полноту взаимодействий в тестируемой реализации

системы. С другой стороны, результаты выполнения интеграционных тестов - один из основных источников информации для процесса улучшения и уточнения архитектуры системы, межмодульных и межкомпонентных интерфейсов. Т.е., с этой точки зрения, интеграционные тесты проверяют корректность взаимодействия компонент системы.

Примером проверки корректности взаимодействия могут служить два модуля, один из которых накапливает сообщения протокола о принятых файлах, а второй выводит этот протокол на экран. В функциональных требованиях к системе записано, что сообщения должны выводиться в обратном хронологическом порядке. Однако, модуль хранения сообщений сохраняет их в прямом порядке, а модуль вывода использует стек для вывода в обратном порядке. Модульные тесты, затрагивающие каждый модуль по отдельности, не дадут здесь никакого эффекта - вполне реальна обратная ситуация, при которой сообщения хранятся в обратном порядке, а выводятся с использованием очереди. Обнаружить потенциальную проблему можно только проверив взаимодействие модулей при помощи интеграционных тестов. Ключевым моментом здесь является то, что в обратном хронологическом порядке сообщения выводит система в целом, т.е., проверив модуль вывода и обнаружив, что он выводит сообщения в прямом порядке, мы не сможем гарантировать, что мы обнаружили дефект.

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.

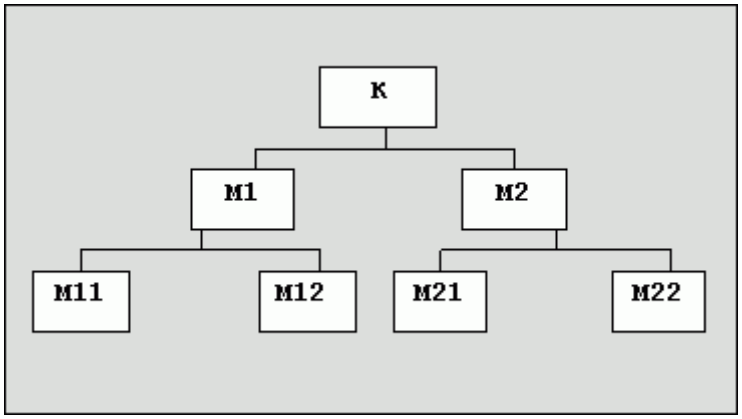


Рис. 5.1. Пример структуры комплекса программ

На Рис. 5.1 приведена структура комплекса программ 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 (K 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 K (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 (html, txt)

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) 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 (html, txt)

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 [5], 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 [10]:

$V(P,C1) = q + k_{in}$, 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]:

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: где 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]:

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.

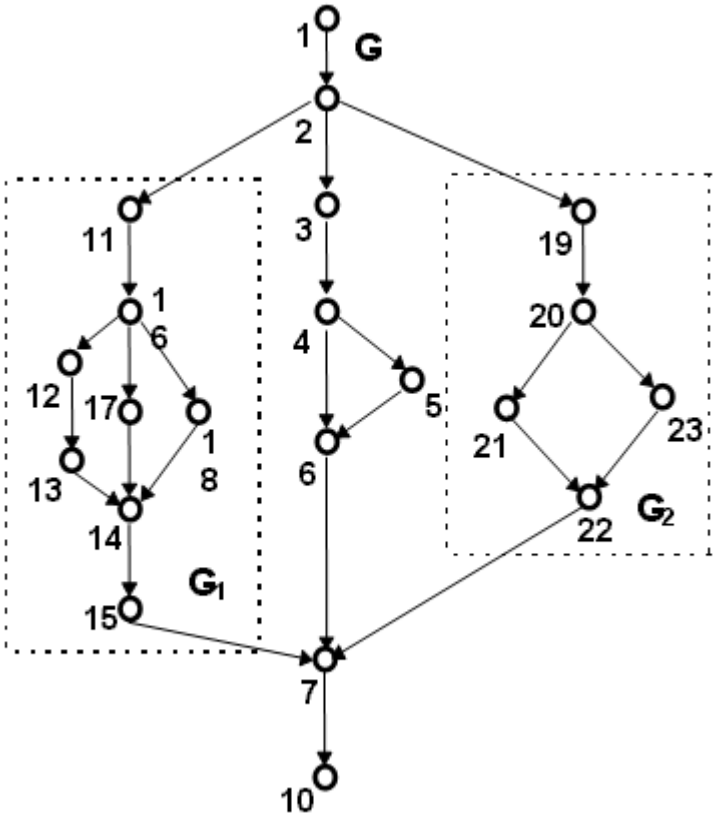


Рис. 4.2. Плоская модель УГП компонента G

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) [19].

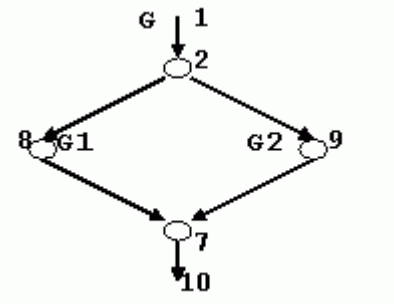


Рис. 5.2. Граф вызовов модулей

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* [20].

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 C#. 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++ [21]".

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, $C1$). В соответствии с введенными выше обозначениями, назовем метод Mod_i, а сложность тестирования - $V(Mod_i, C)$. 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 [22], 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 ***P-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)* [23]. *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. 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 можно выделить четыре *ММ-пути* (1-4) и один *Р-путь* (5):

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

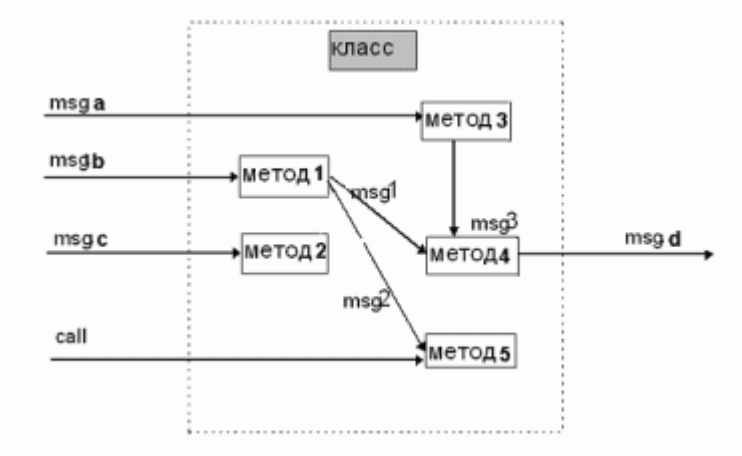


Рис. 6.1. Пример ММ-путей и Р-путей в графовой модели класса

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 M 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: $V(Cls, C) = f(K_{msg}, K_{em})$

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, сложность *интеграционного тестирования* $V(Cls,C)=5$ (множество неизбыточных тестов T 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

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

$$V(P,C) = \Sigma V(Cls_i, C)$$

Формальным представлением описанного выше подхода к тестированию программного проекта служит ***классовая модель программного проекта***, состоящая из дерева классов проекта рис. 6.3 и модели каждого класса, входящего в программный проект рис. 6.4.

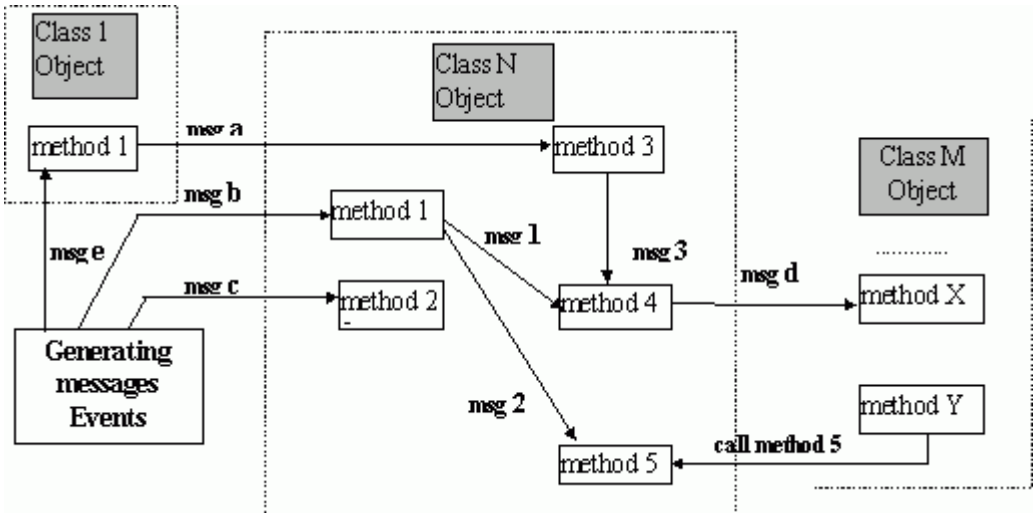


Рис. 6.2. Пример включения объекта в модель программного проекта, построенного с использованием ММ-путей и Р-путей

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.

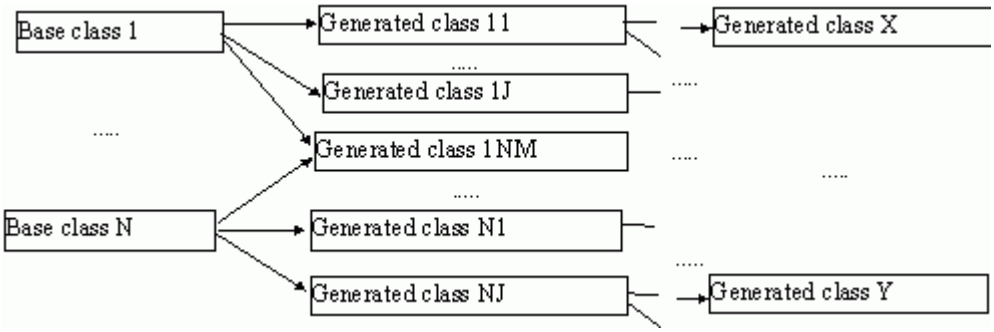


Рис. 6.3. Дерево классов проекта

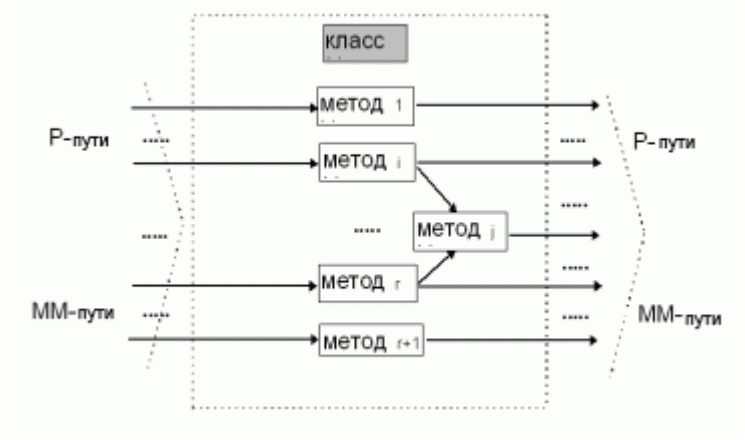


Рис. 6.4. Модель класса, входящего в программный проект

Методика проведения тестирования программы, представленной в виде *классовой модели* программного проекта, включает в себя несколько этапов, соответствующих уровням тестирования рис. 6.5:

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) [23]. 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 [24].

Объектно-ориентированный подход, ставший в настоящее время неявным стандартом разработки программных комплексов, позволяет широко использовать *иерархическую модель* программного проекта, приведенная на рис. 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 С. 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.

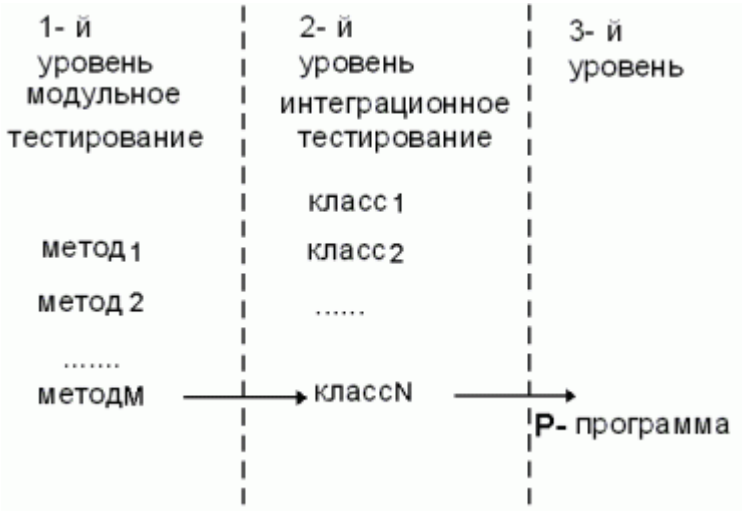


Рис. 6.5. Уровни тестирования классовой модели программного проекта

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`.

Пример интеграционного тестирования

Продемонстрируем тестирование взаимодействий на примере взаимодействия класса `TCommandQueue` и класса `TCommand`, а также, как и при модульном тестировании, разработаем спецификацию тестового случая таблица 6.2:

Таблица 6.2. Спецификация тестового случая для *интеграционного тестирования*

Названия взаимодействующих классов:TCommandQueue, TCommand	**Название теста**:TCommandQueueTest1
Описание теста:тест проверяет возможность создания объекта	

типа TCommand и добавления его в очередь при вызове метода AddCommand

Начальные условия:очередь команд пуста

Ожидаемый результат:в очередь будет добавлена одна команда

На основе этой спецификации разработан тестовый драйвер пример 6.1 - класс `TCommandQueueTester`, который наследуется от класса *`Tester`*.

Класс содержит:

- конструктор, в котором создаются объекты классов `TStore`, `TTerminalBearing` и объект типа `TCommandQueue`
- Методы, реализующие тесты. Каждый тест реализован в отдельном методе.
- Метод `Run`, в котором вызываются методы тестов.
- Метод `dump`, который сохраняет в Log-файле теста информацию обо всех командах, находящихся в очереди в формате - Номер позиции в очереди: полное название команды
- Точку входа в программу - метод `Main`, в котором происходит создание экземпляра класса `TCommandQueueTester`.

```
public TCommandQueueTester()
{
  TB = new TTerminalBearing();
  S = new TStore();
  CommandQueue=new TCommandQueue(S,TB);
  S.CommandQueue=CommandQueue;
...
}
```

Пример 6.1. Объект типа TCommandQueue (html, txt)

```
TCommandQueueTester::TCommandQueueTester()
{
  TB = new TTerminalBearing();
  S = new TStore();
  CommandQueue=new TCommandQueue(S,TB);
  S->CommandQueue=CommandQueue;
}
```

Пример 6.1.1. Объект типа TcommandQueue (C++) (html, txt)

Теперь создадим тест, который проверяет, создается ли объект типа `TCommand`, и добавляется ли команда в конец очереди.

```
private void TCommandQueueTest1()
{
  LogMessage("/////  TCommandQueue Test1  /////");
  LogMessage("Проверяем, создается ли
              объект типа TCommand");
  // В очереди нет команд
  dump();
  // Добавляем команду
  // параметр = -1 означает, что команда
  // должна быть добавлена в конец очереди
CommandQueue.AddCommand(TCommand.GetR,0,0,0,
  new TBearingParam(),new TAxleParam(),-1);
  LogMessage("Command added");
  // В очереди одна команда
  dump();
}
```

Пример 6.2. Тест (html, txt)

```
void TCommandQueueTester::TCommandQueueTest1()
```

```
{
  LogMessage("/////  TCommandQueue Test1  /////");
  LogMessage("Проверяем, создается ли
                объект типа TCommand");
  // В очереди нет команд
  dump();
  // Добавляем команду
  // параметр = -1 означает, что команда
  // должна быть добавлена в конец очереди
  CommandQueue.AddCommand(GetR,0,0,0,
      new TBearingParam(),
      new TAxleParam(),-1);
  LogMessage("Command added");
  // В очереди одна команда
  dump();
}
```

Пример 6.2.1. Тест (C++) (html, txt)

В класс включены еще два разработанных теста.

После завершения теста следует просмотреть текстовый журнал теста, чтобы сравнить полученные результаты с ожидаемыми результатами, заданными в спецификации тестового случая `TCommandQueueTest1` пример 6.3.

```
/////    TCommandQueue Test1  /////
Проверяем, создается ли объект типа TCommand
0 commands in command queue
Command added
1 commands in command queue
0:  ПОЛУЧИТЬ ИЗ ВХОДНОЙ ЯЧЕЙКИ
```

Пример 6.3. Спецификация результатов теста (html, txt)

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* [20].

В результате проведения интеграционного тестирования и устранения всех выявленных дефектов получается согласованная и целостная архитектура программной системы, т.е. можно считать, что интеграционное тестирование - это тестирование архитектуры и низкоуровневых функциональных требований.

Интеграционное тестирование, как правило, представляет собой итеративный процесс, при котором проверяется функциональной все более и более увеличивающейся в размерах совокупности модулей.

Восходящее тестирование. При использовании этого метода подразумевается, что сначала тестируются все программные модули, входящие в состав системы и только затем они объединяются для интеграционного тестирования. При таком подходе значительно упрощается локализация ошибок: если модули протестированы по отдельности, то ошибка при их совместной работе есть проблема их интерфейса. При таком подходе область поиска проблем у тестировщика достаточно узка, и поэтому гораздо выше вероятность правильно идентифицировать дефект.

Нисходящее тестирование предполагает, что процесс интеграционного тестирования движется следом за разработкой. Сначала тестируют только самый верхний управляющий уровень системы, без модулей более низкого уровня. Затем постепенно с более высокоуровневыми модулями интегрируются более низкоуровневые. В результате

применения такого метода отпадает необходимость в драйверах (роль драйвера выполняет более высокоуровневый модуль системы), однако сохраняется нужда в заглушках.

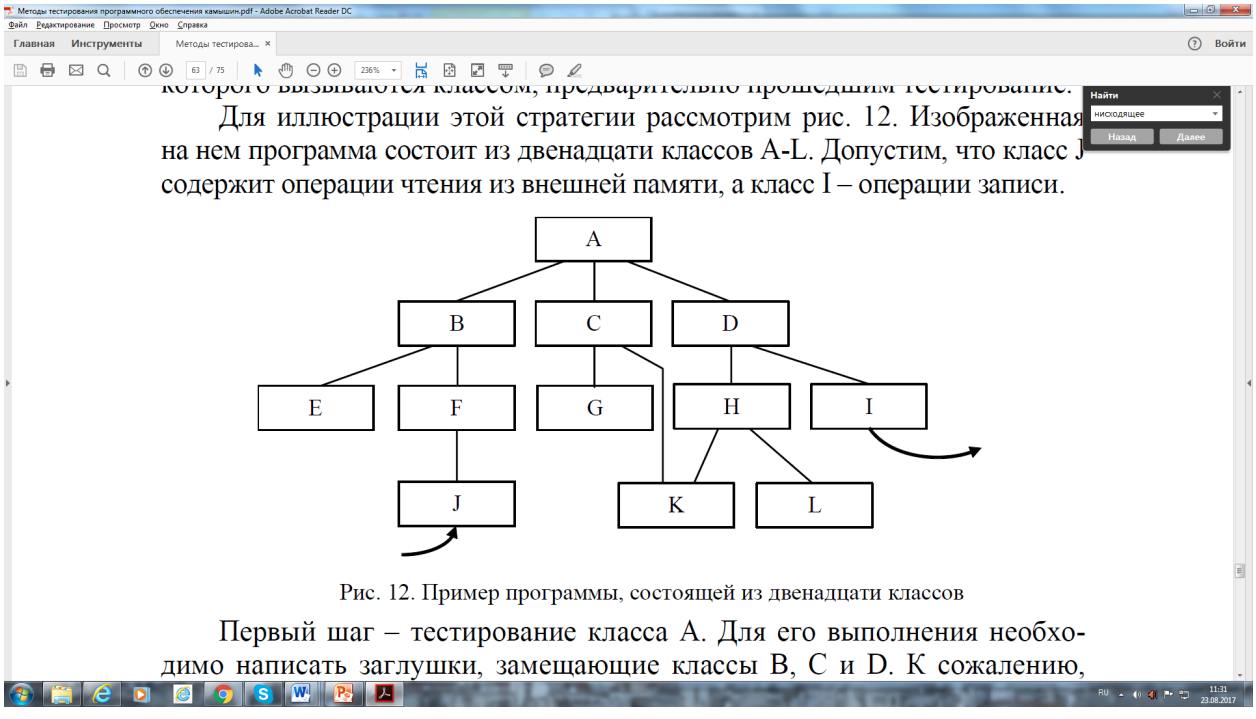
Исследуем две возможные стратегии тестирования: нисходящее и восходящее. Прежде всего внесем ясность в терминологию. Во-первых, термины «нисходящее тестирование», «нисходящая разработка», «нисходящее проектирование» часто используются как синонимы. Действительно, два первых термина являются синонимами (в том смысле, что они подразумевают определенную стратегию при тестировании и создании классов/модулей), но нисходящее проектирование – это совершенно иной и независимый процесс. Программа, спроектированная нисходящим методом, может тестироваться и нисходящим, и восходящим методами. Во-вторых, восходящая разработка или тестирование часто отождествляется с монолитным тестированием. Это недоразумение возникает из-за того, что начало восходящего тестирования идентично монолитному при тестировании нижних или терминальных классов/модулей. Рассмотрим различие между нисходящей и восходящей стратегиями.

3.4.1. Нисходящее тестирование

Нисходящее тестирование начинается с верхнего, головного класса (или модуля) программы. Строгой, корректной процедуры подключения очередного последовательно тестируемого класса не существует. Единственное правило, которым следует руководствоваться при выборе очередного класса, состоит в том, что им должен быть один из классов, методы которого вызываются классом, предварительно прошедшим тестирование. Для иллюстрации этой стратегии рассмотрим рис. 12. Изображенная на нем программа состоит из двенадцати классов A-L. Допустим, что класс J содержит операции чтения из внешней памяти, а класс I – операции записи.

A
B C D
E F G H I
J K L

Рис. 12. Пример программы, состоящей из двенадцати классов



Первый шаг – тестирование класса A. Для его выполнения необходимо написать заглушки, замещающие классы B, C и D. К сожалению, часто неверно понимают функции, выполняемые заглушками. Так, порой можно услышать, что «заглушка» должна только выполнять запись сообщения, устанавливающего: «класс подключен» или «достаточно, чтобы заглушка существовала, не выполняя никакой работы вообще». В большинстве случаев эти утверждения ошибочны. Когда класс A вызывает метод класса B, A предполагает, что B выполняет некую работу, т. е. класс A получает результаты работы метода класса B (например, в форме значений выходных переменных). Когда же метод класса B просто возвращает управление или выдает сообщение об ошибке без передачи в класс A определенных осмысленных результатов, класс A работает неверно не вследствие ошибок в самом классе, а из-за несоответствия ему заглушки. Более того, результат может оказаться неудовлетворительным, если ответ заглушки не меняется в зависимости от условий теста. Например, допустим, что нужно написать заглушку, замещающую программу вычисления квадратного корня, программу поиска в таблице или программу чтения соответствующей записи. Если заглушка всегда возвращает один и тот же фиксированный результат вместо конкретного значения, предполагаемого вызывающим методом класса именно в этом вызове, то вызывающий метод сработает как ошибочный (например, зациклится) или выдаст неверное выходное значение. Следовательно, создание заглушек – задача нетривиальная.

При обсуждении метода нисходящего тестирования часто упускают еще одно положение, а именно форму представления тестов в программе. В нашем примере вопрос состоит в том, как тесты должны быть переданы классу A? Ответ на этот вопрос не является совершенно очевидным, поскольку верхний класс в типичной программе сам не получает входных данных и не выполняет операций ввода-вывода. В верхний класс (в нашем случае, A) данные передаются через одну или несколько заглушек. Для иллюстрации допустим, что классы B, C и D выполняют следующие функции:

B – получает сводку о вспомогательном файле;

C – определяет, соответствует ли недельное положение дел установленному уровню;

D – формирует итоговый отчет за неделю.

В таком случае тестом для класса A является сводка о вспомогательном файле, получаемая от заглушки B. Заглушка D содержит операторы, выдающие ее входные данные на печатающее устройство или дисплей, чтобы сделать возможным анализ результатов прохождения каждого теста.

С этой программой связана еще одна проблема. Поскольку метод класса A вызывает класс B, вероятно, один раз, нужно решить, каким образом передать в A несколько тестов. Одно из решений состоит в том, чтобы вместо B сделать несколько версий заглушки, каждая из которых имеет один фиксированный набор тестовых данных. Тогда для использования любого тестового набора нужно несколько раз исполнить программу, причем всякий раз с новой версией заглушки, замещающей B. Другой вариант решения – записать наборы тестов в файл, заглушкой читать их и передавать в класс A. В общем случае создание заглушки может быть более сложной задачей, чем в разобранном выше примере. Кроме того, часто из-за характеристик программы оказывается необходимым

сообщать тестируемому классу данные от нескольких заглушек, заме-__ щающих классы нижнего уровня; например, класс может получать данные от нескольких вызываемых им методов других классов.
После завершения тестирования класса A одна из заглушек заменяется реальным классом и добавляются заглушки, необходимые уже этому классу. Например, на рис. 13 представлена следующая версия программы.

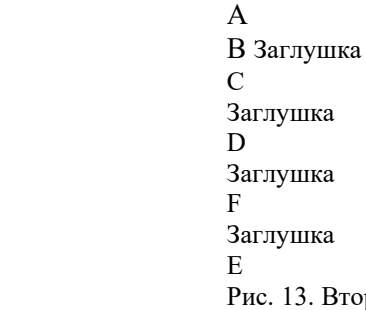


Рис. 13. Второй шаг при нисходящем тестировании

После тестирования верхнего (головного) класса тестирование выполняется в различных последовательностях. Так, если последовательно тестируются все классы, то возможны следующие варианты:
A B C D E F G H I J K L
A B E F J C G K D H L I
A D H I K L C G B F J E
A B F J D I E C G K H L
При параллельном выполнении тестирования могут встречаться иные последовательности. Например, после тестирования класса A одним программистом может тестироваться последовательность A–B, другим – A–C, третьим – A–D. В принципе нет такой последовательности, которой бы отдавалось предпочтение, но рекомендуется придерживаться двух основных правил:
1. Если в программе есть критические в каком-либо смысле части (возможно, класс G), то целесообразно выбирать последовательность, которая включала бы эти части как можно раньше. Критическими могут быть сложный класс, класс с новым алгоритмом или класс со значительным числом предполагаемых ошибок (класс, склонный к ошибкам).
2. Классы, включающие операции ввода-вывода, также необходимо подключать в последовательность тестирования как можно раньше.
Целесообразность первого правила очевидна, второе же следует обсудить дополнительно. Напомним, что при проектировании заглушек возникает проблема, заключающаяся в том, что одни из них должны содержать тесты, а другие – организовывать выдачу результатов на печать или на дисплей. Если к программе подключается реальный класс, содержащий методы ввода, то представление тестов значительно упрощается. Форма их представления становится идентичной той, которая использу- ется в реальной программе для ввода данных (например, из вспомогательного файла или ввод с клавиатуры). Точно так же, если подключаемый класс содержит выходные функции программы, то отпадает необходимость в заглушках, записывающих результаты тестирования. Пусть, например, классы J и I выполняют функции входа-выхода, а G содержит некоторую критическую функцию; тогда пошаговая последовательность может быть следующей:
A B F J D I C G E K H L
и после шестого шага становится такой, как показано на рис. 14.

A
B Заглушка
C
D
F Заглушка
H
Заглушка
E
I
J
Рис. 14. Промежуточное состояние при нисходящем тестировании

По достижении стадии, отражаемой рис. 14, представление тестов и анализ результатов тестирования существенно упрощаются. Появляются дополнительные преимущества. В этот момент уже имеется рабочая версия структуры программы, выполняющая реальные операции ввода-вывода, в то время как часть внутренних функций имитируется заглушками. Эта рабочая версия позволяет выявить ошибки и проблемы, связанные с организацией взаимодействия с человеком; она дает возможность продемонстрировать программу пользователю, вносит ясность в то, что производится испытание всего проекта в целом, а для некоторых является и положительным моральным стимулом. Все это, безусловно, достоинства стратегии нисходящего тестирования.

Однако нисходящее тестирование имеет ряд серьезных недостатков. Пусть состояние проверяемой программы соответствует показанному на рис. 14. На следующем шаге нужно заменить заглушку самим классом H. Для тестирования этого класса требуется спроектировать (или они спроектированы ранее) тесты. Заметим, что все тесты должны быть представлены в виде реальных данных, вводимых через класс J. При этом создаются две проблемы. Во-первых, между классами H и J имеются промежуточные классы (F, B, A и D), поэтому может оказаться *невозможным* пе-

редать методу класса такой текст, который бы соответствовал каждой предварительно описанной ситуации на входе класса H. Во-вторых, даже если есть возможность передать все тесты, то из-за «дистанции» между классом H и точкой ввода в программу возникает довольно трудная интеллектуальная задача – оценить, какими должны быть данные на входе метода класса J, чтобы они соответствовали требуемым тестам класса H. Третья проблема состоит в том, что результаты выполнения теста демонстрируются классом, расположенным довольно далеко от класса, тестируемого в данный момент. Следовательно, установление соответствия между тем, что демонстрируется, и тем, что происходит в классе на самом деле, достаточно сложно, а иногда просто невозможно. Допустим, мы добавляем к схеме рис. 14 класс E. Результаты каждого теста определяются путем анализа выходных результатов методов класса I, но из-за стоящих между классами E и I промежуточных классов трудно восстановить действительные выходные результаты методов класса E (т. е. те результаты, которые он передает в методы класса B).

В нисходящем тестировании в связи с организацией его проведения могут возникнуть еще две проблемы. Некоторые программисты считают, что тестирование может быть совмещено с проектированием программ. Например, если проектируется программа, изображенная на рис. 12, то может сложиться впечатление, что после проектирования двух верхних

уровней следует перейти к кодированию и тестированию классов A и B, C и D и к разработке классов нижнего уровня. Как отмечается в работе [1], такое решение не является разумным. Проектирование программ — процесс итеративный, т. е. при создании классов, занимающих нижний уровень в архитектуре программы, может оказаться необходимым произвести изменения в классах верхнего уровня. Если же классы верхнего уровня уже закодированы и оттестированы, то скорее всего эти изменения внесены не будут, и принятое раньше не лучшее решение получит долгую жизнь.
Последняя проблема заключается в том, что на практике часто переходят к тестированию следующего класса до завершения тестирования предыдущего. Это объясняется двумя причинами: во-первых, трудно вставлять тестовые данные в заглушки и, во-вторых, классы верхнего уровня используют ресурсы классов нижнего уровня. Из рис. 12 видно, что тестирование класса A может потребовать несколько версий заглушки класса B. Программист, тестирующий программу, как правило, решает так: «Я сразу не буду полностью тестировать A – сейчас это трудная задача. Когда подключу класс J, станет легче представлять тесты, и уж тогда я вернусь к тестированию класса A». Конечно, здесь важно только не забыть проверить оставшуюся часть класса тогда, когда это предполагалось сделать. Аналогичная проблема возникает в связи с тем, что классы верхнего уровня также запрашивают ресурсы для использования их классами нижнего уровня (например, открывают файлы). Иногда трудно оп-

ределить, корректно ли эти ресурсы были запрошены (например, верны ли атрибуты открытия файлов) до того момента, пока не начнется тестирование использующих их классов нижнего уровня.

3.4.2. Восходящее тестирование

Рассмотрим восходящую стратегию пошагового тестирования. Во многих отношениях восходящее тестирование противоположно нисходящему; преимущества нисходящего тестирования становятся недостатками восходящего тестирования и, наоборот, недостатки нисходящего тестирования становятся преимуществами восходящего. Имея это в виду, обсудим кратко стратегию восходящего тестирования.
Данная стратегия предполагает начало тестирования с терминальных классов (т. е. классов, не использующих методы других классов). Как и ранее, здесь нет такой процедуры для выбора класса, тестируемого на следующем шаге, которой бы отдавалось предпочтение. Единственное правило состоит в том, чтобы очередной класс использовал уже оттестированные классы.
Если вернуться к рис. 12, то первым шагом должно быть тестирование нескольких или всех классов E, J, G, K, L и I последовательно или параллельно. Для каждого из них требуется свой драйвер, т. е. программа, которая содержит фиксированные тестовые данные, вызывает тестируемый класс и отображает выходные результаты (или сравнивает реальные выходные результаты с ожидаемыми). В отличие от заглушек, драйвер не должен иметь несколько версий, поэтому он может последовательно вызывать тестируемый класс несколько раз. В большинстве случаев драйверы проще разработать, чем заглушки.
Как и в предыдущем случае, на последовательность тестирования влияет критичность природы класса. Если мы решаем, что наиболее критичны

классы D и F, то промежуточное состояние будет соответствовать рис. 15. Следующими шагами могут быть тестирование класса E, затем класса B и комбинирование B с предварительно оттестированными классами E, F, J.

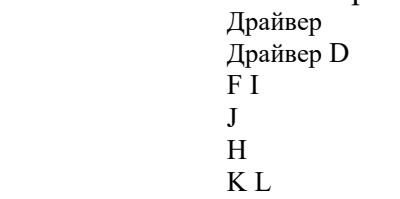


Рис. 15. Промежуточное состояние при восходящем тестировании

Недостаток рассматриваемой стратегии заключается в том, что концепция построения структуры рабочей программы на ранней стадии тестирования отсутствует. Действительно, рабочая программа не существует до тех пор, пока не добавлен последний класс (в примере класс A), и это уже готовая программа. Хотя функции ввода-вывода могут быть проверены прежде, чем собрана вся программа (использовавшиеся классы ввода-вывода показаны на рис. 15), преимущества раннего формирования структуры программы снижаются.

Здесь отсутствуют проблемы, связанные с невозможностью или трудностью создания всех тестовых ситуаций, характерные для нисходящего тестирования. Драйвер как средство тестирования применяется непосредственно к тому классу, который тестируется, нет промежуточных классов, которые следует принимать во внимание. Анализируя другие проблемы, возникающие при нисходящем тестировании, можно заметить, что при восходящем тестировании невозможно принять неразумное решение о совмещении тестирования с проектированием программы, поскольку нельзя начать тестирование до тех пор, пока не спроектированы классы нижнего уровня. Не существует также и трудностей с незавершенностью тестирования одного класса при переходе к тестированию другого, потому что при восходящем тестировании с применением нескольких версий заглушки нет сложностей с представлением тестовых данных.

3.4.3. Сравнение

В табл. 2 показаны относительные недостатки и преимущества нисходящего и восходящего тестирования (за исключением их общих преимуществ как методов пошагового тестирования). Первое преимущество каждого из методов могло бы явиться решающим фактором, однако трудно сказать, где больше недостатков: в классах верхнего уровня или классах нижних уровней типичной программы. Поэтому при выборе стратегии целесообразно взвесить все пункты из табл. 2 с учетом характеристик конкретной программы. Для программы, рассматриваемой в качестве примера, большое значение имеет четвертый из недостатков нисходящего тестирования. Учитывая этот недостаток, а также то, что отладочные средства сокращают потребность в драйверах, но не в заглушках, предпочтение следует отдать стратегии восходящего тестирования.

В заключение отметим, что рассмотренные стратегии нисходящего и восходящего тестирования не являются единственно возможными при пошаговом подходе. В работе [10] рассматриваются еще три варианта стратегии тестирования.

Таблица 2

Сравнение нисходящего и восходящего тестирования

Преимущества	**Недостатки**
Нисходящее тестирование	
1. Имеет преимущества, если ошибки главным образом в верхней части программы. 2. Представление теста облегчается после подключения функции ввода-вывода. 3. Раннее формирование структуры программы позволяет провести ее демонстрацию пользователю и служит моральным стимулом.	1. Необходимо разрабатывать заглушки. 2. Заглушки часто оказываются сложнее, чем кажется вначале. 3. До применения функций ввода-вывода может быть сложно представлять тестовые данные в заглушки. 4. Может оказаться трудным или невозможным создать тестовые условия. 5. Сложнее оценка результатов тестирования. 6. Допускается возможность формирования представления о совмещении тестирования и проектирования. 7. Стимулируется незавершение тестирования некоторых классов/модулей.
Восходящее тестирование	
1. Имеет преимущества, если ошибки главным образом в классе/модуле нижнего уровня. 2. Легче создавать тестовые условия. 3. Проще оценка результатов.	1. Необходимо разрабатывать драйверы. 2. Программа как единое целое не существует до тех пор, пока не добавлен последний класс/модуль.