Г. МАЙЕРС: КОМПОЗИЦИОННОЕ ПРОЕКТИРОВАНИЕ ПРИЛОЖЕНИЯ

27 сентября 2009 г.

АННОТАЦИЯ

Документ представляет собой главу 6 из монографии Г.Майерса, [11] , текст является достаточно близкой копией перевода Ю.Ю.Галимова, под редакцией В.Ш.Кауфмана [3] . Наличие неточностей и лишних толкований объясняется тем, что сначала был выполнен перевод английского текста, а потом получен доступ к переводу Галимова. Данная глава должна принести несомненную практическую пользу. Излагаемая STS-декомпозиция является достаточно понятной и формальной процедурой, позволяющей проектировать структуру приложения, выделяя 'простые' классы и 'простые' связи между ними в случае неинтерактивных приложений, таким образом, что достигается минимальная сложность структуры приложения. Точнее, отдельные функции, выполняемых приложением, распределяются по информационно и функционально прочным модулям и ослабляются связи по данным между модулями посредством использования механизма передачи параметров. В каком-то смысле ее можно рассматривать как аналог процесса нормализации таблиц РСУБД (см. [6]). Далее, как показано в работе [2] процесс декомпозиции интерактивных приложений может быть сведен к STS-декомпозиции Майерса. С методом проектирования отдельно взятого класса можно познакомится в переводе [5] .

Содержание

ПРЕДИСЛОВИЕ

Функция

функция очень часто понимается как обязанность, нечто, что должна выполнить программа.

задача

Словом 'задача' (problem) часто обозначается скорее некоторая деятельность по обработке данных, чем проблему или 'задачу' подлежащую решению.

multiprogramming

мультипрограммирование – представление программы в виде нескольких взаимодействующих процессов, каждый из которых является последовательной программой. Возможно, программирование средов (thread) попадает в эту категорию.

related functions

зависимые друг от друга функции (обязанности).

walk-through

сквозной контроль

модуль

A module is a closed subroutine that can be called from any other module in the program and can be separately compiled (См. стр. 89 [11]) .

absolute storage location

абсолютная ячейка памяти

subroutine

Слово subroutine (subroutine is a portion of code within a larger program, which performs a specific task and is relatively independent of the remaining code. [См. http://en.wikipedia.org/wiki/Subroutine]) будем трактовать как последовательность связанных между собой операторов, выполняющих некоторую функцию и относительно независимых от других частей кода приложения.

Таким образом термин 'модуль', использованный Майерсом, понимается как закрытая (в каком-то смысле) последовательность связанных между собой операторов,

- выполняющих некоторую функцию и относительно независимую от других частей приложения;
- такую, что ее можно вызвать из другой части приложения;
- которая может быть откомпилированна отдельно.

Закрытость означает, что не существует доступа к памяти компьютера внутри модуля иначе, чем через специально предназначенный для этого интерфейс.

Если располагать один класс в одном файле, то, без слишком больших ограничений общности, можно заметить, что, вследствие инкапсуляции и существования публичных методов функций, термин 'класс' удовлетворяет определению термина 'модуль'. То есть, 'класс' есть сужением 'модуль'. Таким образом замена в тексте Г.Майерса слова 'модуль' на слово 'класс' не должна слишком сильно искажать смысл текста.

Обсуждение формальных определений терминов класс и объект можно найти в [4] .

В некоторых случаях, в тексте перевода добавляются примечания с определениями или толкованиями термина из других источников, в основном, из курса лекций ВМиК МГУ 'Технология программирования' [1] .

Далее идет 6 глава Майерса.

ВВЕДЕНИЕ

Следующий процесс проектирования программного обеспечения – проектирование структуры программы. Он включает определение всех модулей системы, их иерархии и сопряжений (интерфейсов) между ними. Если разрабатывается отдельная программа, исходными данными для этого процесса будут детальные внешние спецификации, если же система – детальные внешние спецификации и архитектура системы. В этом последнем случае рассматриваемый процесс состоит в проектировании структуры всех компонент или подсистем полной системы.

Традиционный метод борьбы со сложностью – принцип 'разделяй и властвуй', часто называемый модуляризацией. На практике этот подход часто не приводит к ожидаемому уменьшению сложности. Лисков [10] указала три причины подобной неудачи:

- Модули выполняют слишком много связанных, но различных функций – это делает их логику запутанной;
- При проектировании остались невыявленными общие функции, вследствие чего они рассредоточены (и по - разному реализованы) в разных модулях;
- Модули взаимодействуют посредством совместно используемых или общих данных самым неожиданным образом.

Методология проектирования, называемая композиционным проектированием (composite design) [12] , – это принцип проектирования, рассматриваемый здесь на примере проектирования структуры программы. Композиционное проектированияе состоит, по существу, из двух компонет:

- системы явных проектных оценок, позволяющих решить все три упомянутые проблемы и еще целый ряд дополнительных проблем;
- набор мыслительных процессов, обеспечивающих разбиение программы на множество модулей, их сопряжений и отношений.

В результате композиционного проектирования достигается минимальная сложность структуры программы. Такую программу легче понимать, сопровождать и адаптировать.

НЕЗАВИСИМОСТЬ МОДУЛЕЙ

Чтобы уменьшить сложность программы, нужно разбить ее на множество небольших, в высокой степени независимых друг от друга модулей. Модуль – это замкнутая программа, которую можно вызвать из любого другого модуля в программе и можно компилировать отдельно (отметим, что тем самым исключаются внутренние процедуры PL/1 и параграфы Кобола). Довольно высокой степени независимости можно достичь с помощью двух методов оптимизации:

- усилением внутренних связей в каждом модуле;
- ослаблением взаимосвязи между модулями.

Если рассматривать программу как набор предложений, связанных между собой некоторыми отношениями (как по выполняемым функциям, так и по обрабатываемым данным), то основное, что требуется, – это догадаться, как распределить эти предложения по отдельным 'ящикам' (модулям) так, чтобы предложения внутри каждого модуля были тесно связаны, а связь между любой парой предложений в разных модулях была минимальной. Нужно стремиться, во-первых, реализовывать отдельные функции отдельными модулями (высокая прочность модуля) и ослаблять связь между модулями по данным, применяя формальный механизм передачи параметров (слабое сцепление модулей).

ПРОЧНОСТЬ МОДУЛЯ

Прочность модуля

Прочностью модуля(module strength) будем называть меру его внутренних связей. Чтобы определить прочность модуля, необходимо проанализировать выполняемую им функцию (или функции), с тем чтобы решить, к какому из семи классов он относится. Классы эти специально определены для того, чтобы ввести количественную характеристику 'доброкачественности' конкретных типов модулей.

Перед тем, как мы продолжим, необходимо определить, что понимается под функцией модуля. Модуль имеет три основных атрибута:

- он выполняет одну или больше функций;
- обладает некоторой логикой;
- и используется в одном или нескольких контекстах.

Функция – это внешнее описание модуля; описывается, что делает модуль, когда он вызван, но не как это делается. Логика описывает внутренний алгоритм модуля, другими словами, как модуль выполняет свои функции. Контекст описывает конкретное применение модуля. Например, модуль с функцией 'удалить пробелы из строки символов' может использоваться в контексте 'сжать сообщение для телеобработки'. Чтобы увидеть разницу между функцией и логикой, рассмотрим модуль, функция которого – компилировать программу на PL/1. Он может быть головным модулем 83- модульного компилятора либо быть единственным модулем компилятора. В обоих случаях функция этих двух модулей одинакова, но логика – совершенно разная. Мы видим, что функция модуля может рассматриваться как композиция его логики и функций всех подчиненных (вызываемых им) модулей. Это определение рекурсивно и применяется к любому модулю в иерархии.

Цель проектирования – так определенить модули, чтобы каждый из них выполнял одну функцию (говорят, что такие модули обладают функциональной прочностью). Чтобы понять важность этой цели, ниже рассмотрим семь классов прочности модулей, начиная с самого слабого типа прочности.

Прочный по совпадению (coincindental-strength)

Модуль, прочный по совпадению, – модуль, между операторами и переменными (among its elements) которого нет осмысленных связей. Трудно привести пример такого модуля, так как он не выполняет никаких разумных функций. Описание логики – единственный возможный способ описания модулей этого типа. Одна из причин, по которым такие модули могут возникнуть, – это 'модуляризация' программы post factum, когда мы обнаруживаем одинаковые последовательности команд в нескольких местах программы и решаем сгруппировать их в отдельный модуль. Если эти последовательности (хотя они и кажутся идентичными) имеют разный смысл в тех местах, в которые они первоначально входили, то наш новый модуль является прочным по совпадению. Модуль такого типа слишком тесно связан с вызывающими его модулями, поэтому почти любая его модификация в интересах одного из вызывающих модулей приводит к тому, что для всех остальных он станет работать неправильно.

Модуль прочный по логике (logical-strength)

Модуль, прочный по логике, при каждом вызове выполняет выбранную функцию из набора (функций) связанных с ним. Выбираемая функция обычно запрашивается вызывающим модулем, например с помощью кода функции. Примером может быть модуль, функция которого – читать или писать в файл. Главная проблема с модулями этого типа – это использование одного и того же сопряжения для выполнения многих функций. Это приводит к сложным сопряжениям и неожиданным ошибкам при изменении сопряжения ради одной из функций.

Прочный по классу (classical-strength) модуль

Модуль, прочный по классу, последовательно выполняет набор связанных с ним функций. Самые распространенные примеры – 'начальный' и 'заключительный' модули. Главная проблема с модулями этого типа состоит в том, что обычно они неявно связаны с другими подулями программы, что делает программу трудной для понимания и ведет к ошибкам, когда ее приходится изменять.

замечание

Другое русскоязычное определение – модуль прочный по исполнению: модуль выполняет несколько не связанных функций, отнесенных разработчиком к одному модулю потому, что они необходимы в один и тот же период работы системы.

Процедурно прочный (procedural-strength) модуль

Процедурно прочный модуль последовательно выполняет набор тех связанных с ним функций, которые непосредственно относятся к процедуре решения задачи. Вот пример задачи: написать программу регулирования температуры простого парового котла. В определенной степени подобная задача может определять действия программы. Например, в постановке задачи может быть сказано, что при получении сигнала x следует закрыть клапан y и прочитать и зарегистрировать значение температуры. Модуль с функцией 'закрыть клапан y, прочитать значение температуры парового котла и занести его в журнал' обладает процедурной прочностью. В этом случае единственная проблема, связанная с надежностью, состоит в том, что фрагменты программы, относящиеся к различным функциям, могут быть переплетены. Отметим, что для модулей этого типа, так же как и для большинства других типов, имеются и другие, не связанные с надежностью, проблемы, как показано Майерсом в [12] .

Другое определение процедурной прочности

> Модуль называется процедурно прочным, если он выполняет несколько функций относящихся к одной функциональной процедуре решения задачи. Такой модуль состоит из элементов, реализующих независимые действия, для которых задан порядок работы, то есть порядок передачи управления. Зависимости по данным между элементами нет.

Коммуникационно прочный (communicational-strength) модуль

Коммуникационно прочный модуль – это процедурно прочный модуль с одним дополнительным ограничением: все его функции связаны по данным. Например, модуль 'прочитать следующей запись и обновить главный файл' коммуникационно прочен, так как обе функции связаны между собой тем, что обе работают с одной и той же записью. И здесь обычно возможно переплетение функций, но риск внесения ошибки при модификации несколько меньше, поскольку функции связаны более тесно.

Следующая на нашей шкале – информационная прочность. Однако я ненадолго отложу ее обсуждение.

Функционально прочный (functional-strength) модуль

Функционально прочный модуль – это модуль, выполняющий одну определенную функцию, такую, как 'закрыть клапан у', 'выполнить команду РЕДАКТИРОВАТЬ' или 'подвести итог по сделкам за неделю'. Функциональная прочность – это высшая (лучшая) форма прочности модуля.

Отметим, что функционально прочный модуль может быть описан набором более детальных функций. Например, модуль 'подвести итог по сделкам' можно описать так 'подготовить начальное состояние итоговой таблицы, открыть файл сделок, прочитать читать сделки и обновлять итоговую таблицу'. Глядя на это читатель может подумать, что простой перефразировкой описания модуля понижена его прочность. Однако если эти 'функции более низкого уровня' могут быть рационально описаны как одна хорошо определенная функция 'более высокого уровня', то следует считать, что модуль обладает функциональной прочностью.

Информационно прочный (informational-strength) модуль

Оставшийся тип прочности – информационная прочность. Информационно прочный модуль выполняет несколько функций, причем все они работают с одной и той же структурой данных и каждая представляется собственным входом. Модуль с двумя входами, один из которых соответствует функции

'включить элемент в таблицу символов', а другой – функции 'искать в таблице символов', обладает информационной прочностью. Модуль этого типа может также рассматриваться как физическое объединение нескольких функционально прочных модулей с целью 'упрятывания информации' ('information hiding' [13]), например для того, чтобы укрыть внутри одного модуля все сведения о конкретной структуре данных, ресурсах или устройстве. В упомянутом выше примере вся информация о структуре и расположении таблицы символов скрыта внутри одного модуля. Это имеет то преимущество, что всякий раз, когда удается скрыть некоторый аспект программы внутри одного модуля, независимость ее модулей увеличивается. Упоминавшуюся ранее цель проектирования нужно теперь подправить, чтобы наряду с функционально прочными модулями стремиться к информационно прочным.

Замечание

Информационно прочный модуль – это модуль, выполняющий (реализующий) несколько операций (функций) над одной и той же структурой данных (информационным объектом), которая считается неизвестной вне этого модуля. Для каждой из этих операций в таком модуле имеется свой вход со своей формой обращения к нему. Информационно-прочный модуль может реализовывать, например, абстрактный тип данных.

Цель и положительные эффекты композиционного проектирования

Теперь мы можем переформулировать цель композиционного проектирования следующим образом:

Целью композиционного проектирования

является распределение отдельных функций, выполняемых приложением, по информационно и функционально прочным модулям и минимизация связей по данным между модулями посредством использования метода передачи формальных параметров.

Хотя выше мы сконцентрировали внимание только на связи между прочностью модуля и защищенностью от ошибок, прочность модуля влияет также на адаптируемость программы, трудность тестирования отдельных модулей и степень применимости модуля в других контекстах и других программах ([12]). Шкала прочности упорядочена с учетом всех этих атрибутов.

Отметим, что модуль может соответствовать описанию нескольких типов прочности. Например, коммуникационно прочный модуль удовлетворяет также определению процедурной прочности и прочности по классу. Будем всегда относить модуль к высшему типу прочности, определению которого он удовлетворяет.

СЦЕПЛЕНИЕ МОДУЛЕЙ

Второй важнейший способ увеличить назависимость модулей – ослабить связи между ними. Сцепление модулей, т.е. мера взаимозависимости модулей по данным, характеризуется как способом передачи данных, так и свойствами самих этих данных. Проанализировав любую пару модулей, можно определеить, к какому из шести видов относится сцепление между ними, либо установить, что между ними прямого сцепления нет.

Цель проектирования состоит в определении таких сопряжений между модулями, чтобы все данные передавались между ними в форме явных и простых параметров. Как и раньше, чтобы понять важность этой цели, мы рассмотрим ниже шесть видов сцепления, начиная с самого жесткого (наихудший случай).

Сцепление по содержимому (content coupled)

Два модуля сцеплены по содержимому, если они прямо ссылается на содержимое другого. Например, если модуль A каким - либо образом ссылается на данные модуля B, используя абсолютное смещение, то эти модули сцеплены по содержимому.

Почти любое изменение вносимое в B, или, возможно, просто перекомпиляция модуля B с помощью другой версии комлилятора внесет ошибку в приложение. К счастью, большинство языков программирования высокого уровня делает сцепление по содержимому трудноосуществимым.

Сцепление по общей области (common coupled)

Модули сцеплены по общей области, если они ссылаются на одну и ту же глобальную структуру данных. Модули языка PL/1, которые ссылаются на структуру объявленную как EXTERNAL, сцепленны друг с другом по общей области. Модули FORTRAN-а ссылающиеся на данные в блоке COMMON, и группы модулей, ссылающиеся на абсолютные адреса памяти (включая регистры), также служат примерами сцепления по общей области.

С сцеплением по общей области связан целый ряд проблем. Все такие модули зависят от физичекого упорядочения элементов общей структуры данных, вследствие чего изменение размеров одного элемента данных влияет на все модули. Использование глобальных данных сводит на нет все попытки управлять доступом каждого модуля к данным. Например, в OS/360 фирмы IBM есть большая глобальная структура данных, называемая таблицей вектора связи. Невозможность управлять доступом к этой таблице (и другим глобальным таблицам) привела к многочисленным проблемам в связи с надежностью и адаптируемостью. Имена глобальных переменных связывают модули еще тогда, когда те только создаются. Это значит, что использование

сцепленных по общей области модулей в новых программах затруднено, если не невозможно вообще.

Глобальные данные усложняют и восприятие программы. Рассмотрим следующий код:

```
do while (A);
   call L (X, Y, Z);
   call M (X, Y);
   call N (W, Z);
   call P (Z, X, Y);
end;
```

Если A не является глобальной переменной и если другие неудачные приемы кодирования не используются (такие как совмещение A с W, X, Y или Z), можно констатировать, что цикл никогда не закончится. Если A – глобальная переменная, то сразу нельзя определить, закончится ли выполнение из цикла. Придется исследовать содержимое модулей L, M, N и P, равно как и содержимое всех модулей вызываемых этими четырьмя модулями, чтобы понять только этот цикл DO.

Возражения против использования глобальных данных становятся такими же серьезными, как и возражения против оператора GOTO и этой проблеме уделяется все большее внимание в профессиональной литературе (см. [15] , [14] , [9]).

Сцепление по внешним данным (external coupled)

Модули сцеплены по внешним данным, если они ссылатся на один и тот же глобальный элемент данных (переменную, имеющую единственное поле). Например, модули PL/1, ссылающиеся на одну переменную (но не структуру) объявленную как EXTERNAL, сцепленны друг с другом по внешним данным. Сцепление по внешним данным создает почти все проблемы, свойственные сцеплению по общей области. Однако проблемы зависимости от физического упорядочения элементов в структуре не возникает.

Сцепление по управлению (control coupled)

Модули сцеплены по управлению, если один явно управляет функционированием второго, например, используя код конкретной функции в качестве параметра. Сцепление по управлению и прочность по логике обычно сопутствуют друг другу, поэтому основная проблема здесь та же, что и с прочностью по логике: использование одного и того же (сложного) сопряжения для выполнения многих функций. Сцепление по управлению частно предполагает также, что вызывающий модуль имеет некоторое предствление о логике вызываемого модуля, что уменьшает их независимость.

Сцепление по формату (stamp coupled)

Группа модулей сцеплена по формату, если они ссылаются на одну и ту же неглобальную структуру данных. Если модуль A вызывает модуль B и передает ему (через формальные параметры) запись анкетных данных служащего, и как A, так и B чувствительны к изменению структуры или формата этой записи, то A и B сцеплены по формату.

Сцепления по формату следует, где это возможно избегать, поскольку оно создает ненужные связи между модулями. Предположим, что модулю B нужны только некоторые поля в анкете. Передача ему всей анкеты вынуждает его заниматься ею целиком, таким образом увеличивается вероятность того, что он неумышленно ее изменит. (По-видимому, можно утверждать, что, чем больше посторонних данных доступно модулю, тем больше возможность ошибки.)

Сцепление по формату часто можно исключить, изолируя все функции, работающие с конкретной структурой данных, в информационно прочном модуле. Другим модулям может понадобиться имя этой структуры, но они и знают только это имя (адрес), а не формат. Такая техника будет проиллюстрирована ниже в этой главе.

Сцепление по данным (data coupled)

Два модуля сцеплены по данным, если один вызывает другой и все входные и выходные параметры вызываемого модуля – простые (не структурированные) элементы данных. Предположим, что функция модуля B из предыдущего примера – надпечатать конверт для служащего. Вместо того, чтобы передавать ему всю анкету, мы могли бы передать ему в качестве аргументов фамилию служащего, улицу, дом, город, штат и почтовый индекс. Модуль B теперь не зависит от записи анкетных данных. A и B стали более независимы, и вероятность ошибки в B меньше, поскольку автор модуля B имеет дело с меньшим количеством данных.

Дальнейшее обсуждения сцепления модулей

Как и в случае с прочностью модуля, сцепление модулей влияет и на другие, не рассматриваемые здесь специально свойства программы (на адаптируемость, легкость тестирования, возможность повторного использования модулей, простоту или сложность мультипрограммирования ([12]).

Сцепление пары модулей может удовлетворять определениям нескольких типов. Например, два модуля могут быть сцеплены и по образцу, и по общей области. В этом случае мы относим модуль к самому жесткому (худшему) из этих типов сцепления (в данном случае – сцепление по сцепление по общей области).

Степень прочности и сцепления можно использовать для оценки существующего проекта и как руководящий принцип при проектировании новой программы. Это, однако, не означает, что проект, в котором в отдельных случаях прочность и сцепление далеки от идеала, обязательно плох. Исходя из некоторого компромиссного решения, проектировщик может пойти на включение модуля, прочного все лишь по логике. Однако поступая так, он должен уметь убедительно объяснить причины и хорошо понимать следствия своего компромиссного решения. Это, во всяком случае, лучше, чем проектировать, основываясь только на интуиции и полагаться на счастливый случай.

Высокая прочность и слабое сцепление способствует независимости модулей, поскольку они сводят к минимуму их взаимодействие и их предположения друг о друге. Следующие три критерия проектирования, сформулированные Хольтом в [8] , хорошо подытоживает сказанное.

1. Сложность взаимодействия модуля с другими модулями должна быть меньше сложности его внутренней структуры.
2. Хороший модуль снаружи проще, чем внутри.
3. Хороший модель проще использовать, чем построить.

ДРУГИЕ ХАРАКТЕРИСТИКИ

В дополнение к прочности и сцеплению есть и другие характеристики, оказывающее влияние на независимость модулей. Они кратко охарактеризованы ниже.

Размер модуля

Размер модуля влияет на степень независимости элементов программы, легкости ее чтения и тестирования (например, за счет числа ветвей). Можно было бы удовлетворить критериям высокой прочности и минимального сцепления, спроектировав программу как один огромный модуль, но вряд ли таким образом была бы достигнута высокая степень независимости. Желательно разбивать программу на достаточно большое число модулей, поскольку модули представляют собой явные барьеры внутри программы, что сокращает число связей между операторами и данными программы. Как правило, модуль должен содержать от 10 до 100 выполняемых операторов языка высокого уровня.

Предсказуемые модули

Модуль назвается предсказуемым (рутинным), если его работа не зависит от предыстории его использования. Модуль, хранящий следы своих состояний при последовательных вызовах (например, с помощью установки 'переключателя

для первого раза'), не является предсказуемым. Все модули должны быть предсказуемыми, т.е. они они не должны сохранять никаких 'воспоминаний' о предыдущем вызове. Хитрые, неуловимые, зависящие от времени ошибки возникают в тех программах, которые пытаются многократно вызывать непредсказуемый модуль.

Структура принятия решения (Decision structure)

Всюду, где это возможно, желательно организовать модули и принятие решений в них таким образом, чтобы те модули, на которые прямо влияет принятое решение, были подчиненными (вызываемыми) по отношению к модулю принимающему решение. Таким образом обычно удается исключить передачу специальных параметров - индикаторов, представляющих решения, которые должны быть приняты, а также принимать влияющие на управление программой решения на высоком уровне в иерархии программы.

Минимизация доступа к данным

Объем данных, на которые модуль может ссылаться, должен быть сведен к минимуму. Исключение сцепления по общей области, внешним данным и по формату – крупный шаг в этом направлении. Проектировщик должен попытаться изолировать сведения о любой конкретной структуре или записи в базе данных в отдельном модуле (или небольшом подмножестве модулей), возможно, за счет использования информационно прочных модулей. Проблема глобальных данных не должна решаться передачей одного огромного списка параметров всем модулям. Следуя этим правилам, вы уменьшаете область доступа каждого модуля, благодаря чему ошибки легче изолировать, а область их влияния – уменьшить.

Внутренние процедуры

Внутренняя процедура или подпрограмма – это замкнутая подпрограмма, физически расположенная в вызывающем модуле. Внутренних процедур желательно избегать по нескольким причинам. Внутренние процедуры трудно изолировать для тестирования (автономного тестирования), и они не могут быть вызваны из модулей, отличных от тех, которые их физически содержат. Это не соответствует идее 'повторного использования'. Конечно, имеется альтернативный вариант: включить копии внутренней процедуры во все модули, которым она нужна. Однако это часто приводит к ошибкам (копии одной и той же процедуры часто становятся 'не совсем точными копиями') и усложняет сопровождение программы (когда процедура изменяется, все используемые ее модули должны быть перекомпилированы). Наконец, если только при программировании не установлена строжайшая дисциплина, внутренние процедуры будут иметь плохие сцепления с вызывающими их

модулями. Когда возникает потребность во внутренней процедуре, проектировщик должен рассмотреть возможность оформления ее в виде отдельного модуля.

КОМПОЗИЦИОННЫЙ АНАЛИЗ

Прочность модуля, сцепление и другие рассматривавшиеся характеристики полезны при оценке альтернатив, но они не определяют явно ход мысли при проектировании. В рамках композиционного проектирования имеется процесс, называемый композиционным анализом, – нисходящий процесс продумывания проекта. Композиционный анализ включает анализ структуры задачи и анализ преобразования данных по мере их прохождения сквозь эту структуру. Эта информация используется для разбиения задачи на 'слои' модулей. Каждый модуль рассматривается затем как отдельная подзадача, и анализ повторяется рекурсивно.

Имеются три основные стратегии декомпозиции при применении композиционного анализа.

- Декомпозиция STS (source/transform/sink) предполагает деление задачи на функции, занимающиеся получением данных, изменением их формы и затем доставкой в некоторую точку вне задачи;
- Операционная декомпозиция (transaction decomposition) состоит в делении задачи на функции - 'сестры', каждая из которых выполняет операции отдельного типа;
- Функциональная декомпозиция – это деление задачи на функции, выполняющие преобразование данных.

STS декомпозицию обычно применяют для выделения первого слоя модулей, а затем, к каждой подзадаче применяется отдна из стрех стратегий, причем выбор зависит от характеристик задачи.

Операционная и функциональная декомпозиции – это, в основном, интуитивные процессы, и о них немногое можно добавить к тому, что уже сказано. Напротив, STS декомпозиция – более сложный процесс, и его можно кратко описать в виде следующих шагов:

- Основываясь на потоке данных в задаче, обрисуйте ее структуру в виде 3-10 процессов;
- Определите главный входной поток данных, поступающий в задачу, и главный выходной поток;
- Проследите за изменением входного потока данных при прохождении по структуре задачи. При этом вы обнаружите два явления: входной поток будет изменять форму, становясь все более абстрактнее по мере того, как вы следуете по структуре задачи, и, в конце концов, вы

попадаете в точку, где он как будто исчезает. Точка, где он появляется в последний раз, называется точкой наивысшей абстракции входного потока.

Выполните аналогичный анализ, выходного потока данных, начиная с 'конца' структуры задачи и двигаясь в обратном направлении. Определите точку, где выходной поток впервые появляется в своей самой абстрактной форме;

Эти точки представляют особый интерес, поскольку они делят задачу на наиболее назависимые части (обычно три);

- Представьте эти части как функции и определите модули, выполняющие каждую из этих функций; Эти модули становятся подчиненными по отношению к модулю, декомпозиция которого выполняется.
- Определите сопряжения этих модулей. В этот момент необходимо только определить виды данных для каждого сопряжения. То есть, вам следует дать качественное описание входных и выходных аргументов, не заботясь об их точной природе (порядок, атрибуты, представление). Детали каждого сопряжения будут определены в одном из последующих процессов проектирования (внешнее проектирование модуля, описанное в гл.8) (см. гл. 8 [11]).

Процесс декомпозиции продолжается на следующих, более низких уровнях в иерархии, вплоть до момента остановки. Этот момент определяется по следующему правилу: логика модулей должна стать интуитивно понятной (это, вероятно, означает, что размер модуля не будет превосходить 50 операторов).

Результатом анализа является иерархическая структурная схема, отражающая структурные отношения между всеми модулями (кто кого вызывает), функции каждого модуля и сопряжения между ними. Обозначения для таких схем описаны Майерсом в [12] .

ПРИМЕР КОМПОЗИЦИОННОГО АНАЛИЗА

Слово 'объектный' не имеет ничего общего с объектно - ориентированным программированием. Объектный модуль – файл, который получается на выходе компилятора после компиляции файла с программой на некотором языке программирования. Можно считать, что объектный модуль содержит машинный код.

Простейший путь к пониманию композиционного анализа – рассмотреть его применение на примере. Этот пример будет использоваться в последующих главах для иллюстрации других процессов проектирования и тестирования.

offset	
0	proc M
	.
	.
54	call C
	.
	.
100	entry B
	.
	.
200	y dcl addr(x)
	.
	.
220	x dcl
	.
	.
300	

offset	
0	proc C
	.
	.
34	call B
	.
	.
60	

Рис. 1: Source program (hypotetical language)

Выбирая подходящий пример, я сразу же столкнулся с несколькими проблемами. В качестве примера нельзя взять большую программу; она должна по размерам подходить для учебника, но при этом оставаться нетривиальной. Прикладная программа (например, программа начисления зарплаты) может не заинтересовать системных программистов и даже многих прикладных программистов; привлекательность какой - либо компоненты операционной системы была бы в такой же степени ограниченной. В качестве компромисса я выбрал загрузчик – программу, находящуюся как бы посередине между прикладными программами и операционной системой. Другая причина, по которой я выбрал в качестве примера загрузчик, такова: я надеялся, что большинство читателей будет по крайней мере в общих чертах знакомо с его назначением.

Как уже говорилось в начале главы, исходной информацией для процесса проектирования структуры программы являются детальные внешние спецификации. Вместо того чтобы дать здесь такие спецификации для загрузчика, я просто опишу его назначение, а также входные и выходные данные настолько подробно, чтобы мы могли спроектировать его структуру.

Загрузчик

Назначение загрузчика – поместить программу в основную память в готовом к выполнению состоянии (некоторые загрузчики сразу же и активизируют эту программу, но такие подробности нас здесь не интересуют). Исходными данными для загрузчика является файл (далее обозначаемый как INFILE),

содержащий один или несколько объектных модулей, выработанных компилятором. Загрузчик может иметь в качестве исходных данных также и файл библиотечных программ PROGLIB, содержащий большую библиотеку объектных модулей, которые могут потребоваться в загружаемой программе. Результатами работы загрузчика являются: загруженная в основную память программа и выходной файл OUTFILE, который содержит таблицу загрузки, описывающую расположение модулей загруженной программы в оперативной памяти, и список сообщений об ошибках, если они есть.

ESD	M	MD	0000	
ESD	B	EP	0100	
ESD	C	ER	0D00	
TXT			0300	
TXT	———-	000000	———-	adcon at offset 54 to point to module C
TXT	———-	000220	———-	adcon at offest 200 to point to X
RLD	0054	3		
RLD	0200	1		
END				
ESD	C	MD	0000	
ESD	B	ER	0000	
TXT			0060	
TXT	———-	000000 —	———-	adcon at offset 34 to point to entry B
RLD	0034	2		
END				

Рис. 2: INFILE contents

INFILE – это последовательный файл, содержащий один или несколько объектных модулей. PROGLIB – некоторый файл с индексным или библиотечным методом доступа, в котором объектные модули хранятся раздельно. Все они имеют одинаковый формат: одна или несколько записей ESD (external symbol dictionary), описывающих внешние имена и внешние ссылки в этом модуле, две или больше текстовых записей TXT (text), которые содержат объктный (машинный) код модуля, за ними может следовать несколько записей словаря перемещений RLD (relocation dictionary), описывающих все адресные константы модуля и (обязательно) одна END запись.

Каждая запись содержит некоторое символическое имя, его тип (имя модуля MD, имя точки входа EP или ссылка на внешнее имя ER) и относительное смещение внутри модуля.

Первая запись TXT содержит размер объектного модуля. Каждая следующая TXT-запись содержит размер объектного кода и поле, указывающее его длину в этой записи.

Записи RLD описывает все адресные константы в модуле, т.е. адреса, которые должны быть настроены, когда модуль получает определенное

```
   OUTFILE  Output
Loader Memory Map
   Module/entry pt.       Location      Type
        M                      100000      mod
        B                      100100      ep
        C                      100300      mod
Main Storage  Output
  Locations 100000 - 1002FF contain code for module M
    location 100054 contains 100300
    location 100200 contains 100220
  Locations 100300 - 1003F6 contain code for module C
    location 100334 contains 100100
```

Рис. 3: Corresponding loader output

место в оперативной памяти. RLD - запись содержит относительное смещение адресной константы и номер соответствующей записи ESD (каждая адресная константа в объектном модуле связана с внешним именем). Адресным константам, указывающим на области внутри модуля, соответствует запись ESD для имени входа. Адресным константам, указывающим на другие модули, соответствуют записи ESD для ссылок на внешние имена других модулей.

Загрузить программу – это значит сделать следующее:

- загрузить объектный код каждого модуля из INFILE на предназначенное для него место в оперативной памяти;
- Проверить, что все внешние ссылки разрешены. Например, если загружаемый модуль A содержит вызов подпрограммы из модуля B, то загрузчик должен убедиться, что модуль B также загружен. Если модуль B отсутствует в INFILE, он ищется в PROGLIB и, в случае удачи, загружается оттуда. Отметим, что модуль B может тоже содержать внешние ссылки на другие модули, которые необходимо разрешить.
- Настроить все адресные константы. Адресная константа – это часть объектного кода, содержащего некоторый адрес. Мы будем предполагать, что адресные константы имеют одну фиксированную длину. Компилятор не имеет информации о том, в какое место оперативной памяти будет загружена программа, таким образом он помещает в поля адресных констант относительные адреса. Для каждой константы, указывающей на память внутри того же модуля, компилятор помещает смещение этой памяти относительно начала модуля. Адресным константам, содержащих внешние ссылки, компилятором присваивается значение 0. Как только все необходимые модули оказываются загружены в оперативную память, загрузчик присваивает соответствующие значения всем адресным константам.

Вместо того, чтобы объяснить этот процесс загрузки более подробно, я привел на схемах 1 и 2 образец исходных данных загрузчика, а соответствующий

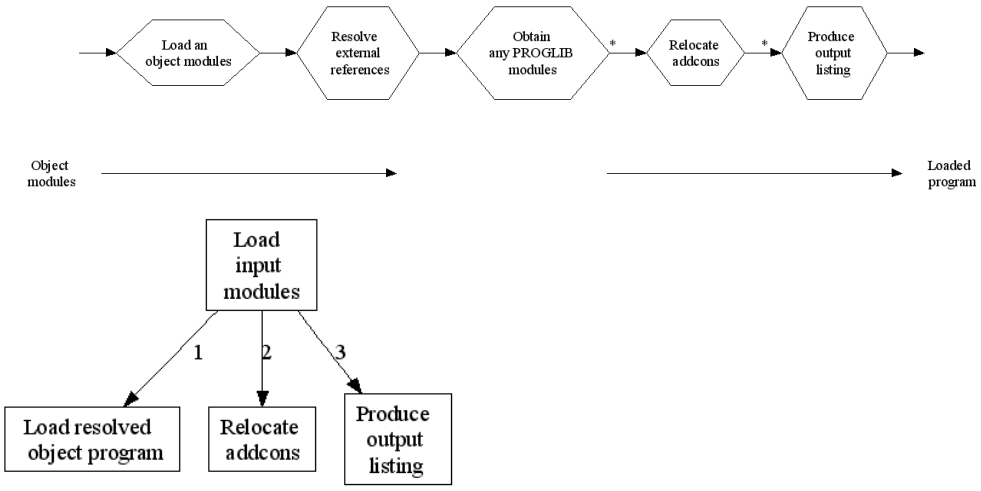
ему результат на схеме 3 . Изучив схемы 1 , 2 и 3 можно получить достаточно информации для понимания задачи проектирования загрузчика. Кроме того, мы предполагаем что загрузчик выполняется в некоторой операционной системе, имеющей средства распределения оперативной памяти и средства ввода/вывода, необходимые для выполнения операций чтения/записи/поиска во входных и выходных файлах.

Первый шаг в проектировании загрузчика состоит в определении главный модуля, функция которого эквивалентна функции всего загрузчика. Назовем этот модуль LOAD-A-PROGRAM.

На следующем шаге проектирования надо рассмотреть этот модуль как задачу, подлежащую решению, и, используя STS декомпозицию, разложить его на более мелкие функции. На схеме 4 изображено какие пять процессов выделяются в нашей задаче, если проследить за потоком данных. Основной входной поток – это объектные модули; основной результат – загруженная программа (таблица загрузки представляет собой вторичный результат). Обратив внимание на то, что объектный модуль является модулем с перемещаемыми адресными константами, а загруженная программа – множеством модулей с абсолютными значениями адресными констант, легко найти точки наивысшей абстракции. Они на схеме обозначены звездочками. Исходная задача теперь разделена на три функции и начата разработка структуры модуля, как показано на схеме 4 .

На этом шаге проектирования осталось определить только интерфейсы модулей (отмеченные цифрами 1, 2, 3). Начнем с интерфейса 3. Мы знаем, что выходной листинг состоит из таблицы загрузки (списка внешних имен и их абсолютные адреса) и сообщений об ошибках. Входными параметрами интерфейса 3 будут MSGLST – список сообщений об ошибках и ESTAB – таблица, содержащая внешние имена, их типы и абсолютные адреса. Заметим, что мы не интересуемся определением точного представления данных на протяжении процесса проектирования структуры программы (таким образом может быть позднее определена как массив или односвязный список). Выходные данные интерфейса 3 – код ошибки EC, который будет определен позже.

Для функции RELOCATE-ADCONS необходимы входные параметры двух типов: описание внешних имен (точнее их абсолютные адреса), и описание всех адресных констант. Таким образом интерфейс 2 имеет на входе таблицу ESTAB и RLTAB, таблицу содержащую указатель на каждую адресную константу и указатель на соответствующий элемент таблицы ESTAB (или его индекс в таблице). Нет необходимости передавать загруженные объектные модули как параметры, так как ESTAB содержит указатели на них. На выходе интерфейс 2 имеет только код ошибки. Теперь определен и интерфейса 1; он не имеет входных данных, выходными будут ESTAB, RLTAB и список сообщений об ошибках – MSGLIST, обнаруженных при выполнении функции.



ESTAB:

name	type	machine address

RLTAB:

ESTAB entry number	machine address of adcon

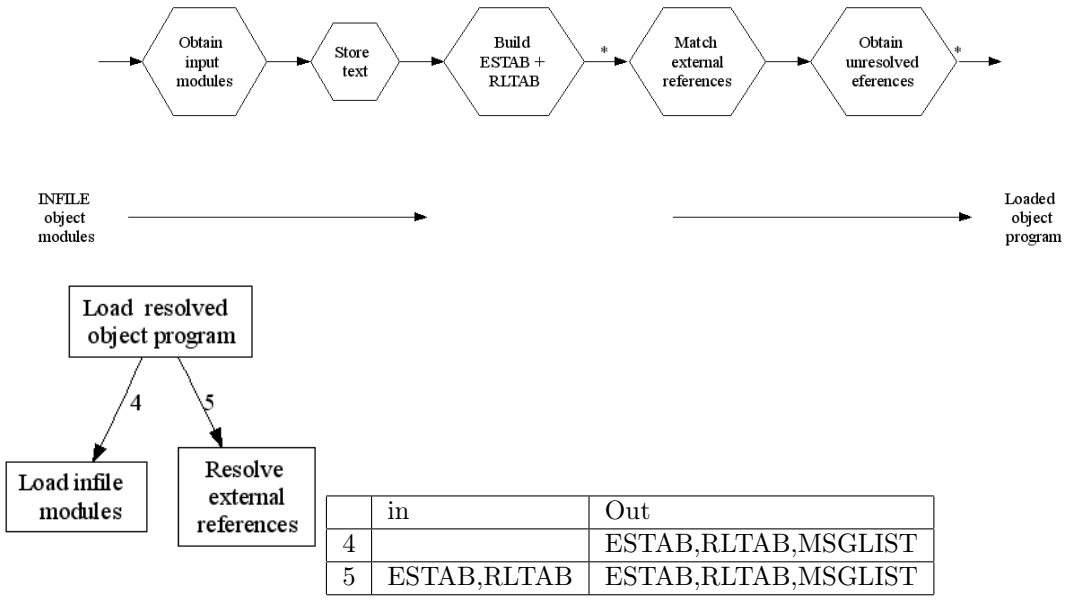
INTERFACES:

	in	out
1		ESTAB,RLTAB,MSGLIST
2	ESTAB,RLTAB	EC
3	ESTAB,MSGLIST	EC

Рис. 4: Начальная декомпозиция

На следующем шаге нужно взять модули, изображенные на схеме 4 и заняться их декомпозицией. Логику модулей RELOCATE-ADCONS и PRODUCE-OUTPUT-LISTING легко себе представить, поэтому их декомпозицией мы здесь заниматься не будем. Нам остается только модуль LOAD-RESOLVED-OBJECT-PROGRAM. Вначале посмотрим на модуль, как на задачу для решения и набросаем его структуру как показано на схеме 5 . Входной поток – множество объектных модулей из INFILE, а выходной поток – объектная программа, в которой замкнуты все внешние ссылки. Точка наивысшей абстракции входного потока является место, где все модули из INFILE размещены в оперативной памяти и занесены в таблицы ESTAB и RLTAB. Выходной поток появляется впервые только на самом выходе задачи. Эти две точки разделяют задачу на две функци и, следовательно, определяют два подчиненных модуля.

Интерфейс 4 не имеет входных параметров и три типа выходных: ES-



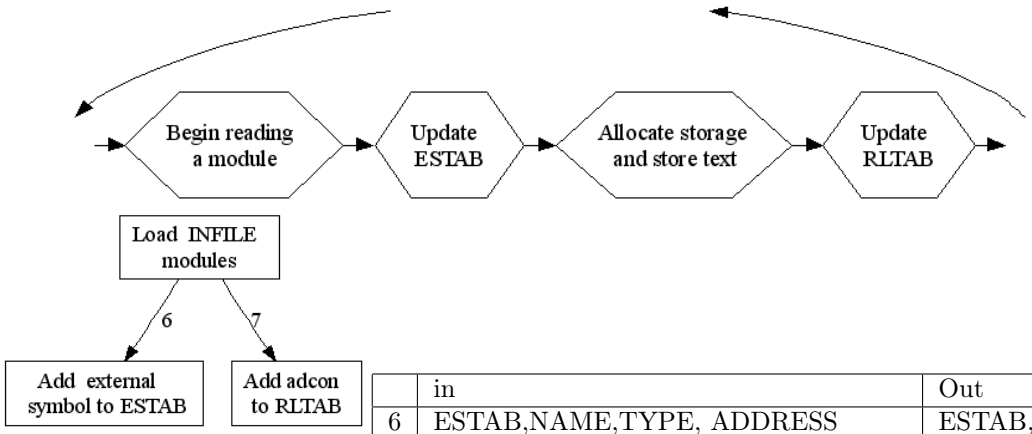
	in	Out
4		ESTAB,RLTAB,MSGLIST
5	ESTAB,RLTAB	ESTAB,RLTAB,MSGLIST

Рис. 5: Декомпозиция LOAD-RESOLVED-OBJECT-PROGRAM

TAB, RLTAB и список сообщений об ошибках. Интерфейсу 5 очевидно необходима ESTAB в качестве входного параметра. Однако, так как модуль RESOLVE-EXTERNAL-SYMBOLS может загружать модули из PROGLIB, он должен иметь возможность добавлять записи в обе таблицы ESTAB и RLTAB и, следовательно, они ему требуются в качестве входных и выходных параметров.

Двигаясь вниз по иерархии, мы можем выполнить декомпозицию модуля LOAD-INFILE-MODULES. Эта задача по своей структуре является циклическим процессом, изображенным на схеме 6 . Задача достаточно простая и в дальнейшей декомпозиции не нуждается, но для целей минимизации количества модулей, которые имеют доступ к обоим таблицам ESTAB и RLTAB , в ней выделены две функции. Заметим, что LOAD-INFILE-MODULES так же обращается к операционной системе для чтения из INFILE и выделения блоков оперативной памяти. ESTAB будет входным параметром интерфейса 6, а RLTAB – входным параметром интерфейса 7, что позволяет обоим модулям быть предсказуемыми и повторно входимыми. Одним из значений EC (error code) в интерфейсе 6 будет сообщать о попытке ввести дублируемое имя с типом MD или EP в таблицу ESTAB.

Далее для декомпозиции выберем модуль RESOLVE-EXTERNAL-REFERENCES из схемы 5 . Входной поток – множество (возможно, пустое) неразрешенных внешних имен, а выходной поток – готовая для загрузки в оперативную память программа. Точки высшей абстракции и результат декомпозиции указаны на схеме 7 .



	in	Out
6	ESTAB,NAME,TYPE, ADDRESS	ESTAB,ENTRY NO, EC
7	RLTAB,ENTRY NO, ADCON ADRESS	RLTAB,EC

Рис. 6: Декомпозиция LOAD-INFILE-MODULES

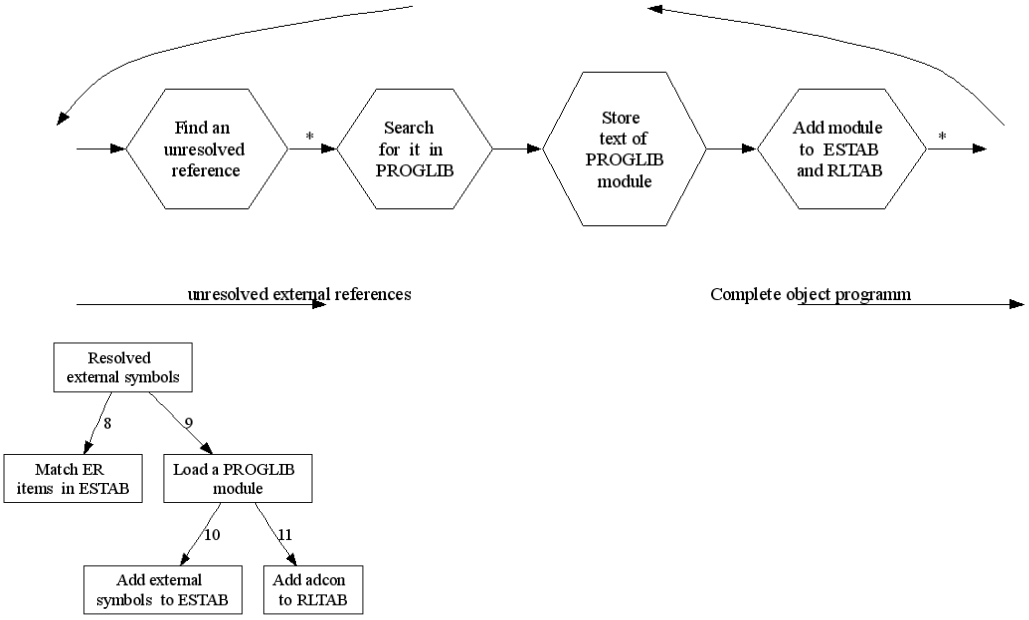
В этом месте мы стоим перед интересным компромиссом. Модули LOAD-PROGLIB-MODULE и LOAD-INFILE-MODULE очень похожи своими функциями, поэтому рекомендуется сделать их обобщение, которое сможет выполнять обе функции. Я же выбрал проектирование двух простых модулей (хотя это не слишком хорошее решение) по следующим причинам:

- файлы имеет различную структуру;
- использование одного модуля повлечет сцепление по управлению (одному модулю нужно было бы явно указывать файл, из котого следует читать).

Отметим, однако, что я могу использовать подчиненные модули LOAD-INFILE-MODULES как показано на схеме.

Модуль MATCH-ER-ITEMS достаточно простой. Он находит неразрешенные имена в ESTAB (они отмечаются нулем в поле адреса). Если встречается неразрешенное имя, модуль находит вхождение в ESTAB элемента с типом MD или EP с таким же именем. Если находит, то переносит его адрес в поле адреса неразрешенного имени в ESTAB и продолжает поиск неразрешенных имен. Если не осталось неразрешенных имен он возвращает DONEFLAG или, если встречено имя, которое не удается разрешить, возвращает найденное имя.

Модуль RESOLVE-EXTERNAL-SYMBOLS тоже достаточно простой. Он циклически вызывает MATCH-ER-ITEMS и LOAD-PROGLIB-MODULE пока не разрешит все имена или не найдет все неразрешенные имена в PROGLIB. Это не самый эффективный, зато самый простой метод для выполнения заданной функции. Как писал Кнут 'Преждевременная оптимизация – корень



	in	Out
8	ESTAB	ESTAB, DONEFLAG, NAME OF AN UNRESOLVERD ER
9	ESTAB, RLTAB, MODULE NAME	ESTAB, RLTAB, MSGLIST
10	ESTAB,NAME,TYPE, ADDRES	ESTAB,ENTRY NO, EC
11	RLTAB,ENTRY NO, ADCON ADRESS	RLTAB,EC

Рис. 7: Последняя декомпозиция

всех проблем' (Premature optimization is the root of all evil. см. [7]) После получения работающего приложения, можно произвести измерения производительности и оптимизировать логику одного или больше модулей (Эти подробности упоминаются здесь только для того, чтобы читатель понимал особенности нашего загрузчика; в действительности в этот не следует интересоваться логикой модуля).

Оценивая текущее состояние нашего проектирования, я больше не вижу модулей, подлежащих разбиению. Можно было бы поэтому объявить проектирование законченным. Однако я по-прежнему имею в виду упоминавшуюся ранее цель – свести к минимуму число модулей, знающих свойства и представление ESTAB и RLTAB. Я могу добиться этого, создавая из некоторых модулей информационно прочные обобщенные модули, как это показано на схеме 8 . Чтобы изолировать знания об ESTAB в одном модуле, мне нужно создать новую функцию (вход): FIND-ITEM-IN-ESTAB, которую вызывает

RELOCATE-ADCONS и PROCEDURE-OUTPUT-LISTING. Заметим, что хотя другие модули передают ESTAB и RLTAB как параметры, только два информационно прочных модуля ESTABMNGR и RLTABMNGR имеют доступ к 'внутренностям' таблиц. Например, ESTABMNGR знает о формате записей ESTAB, знает является ли ESTAB просто последовательная таблицей или списком, и является она отсортированной или нет.

В окончательном проекте, изображенном на схеме 8 мы достигли следующих результатов:

- Шесть модулей являются функционально прочными, два оставшихся имеют информационную прочность;
- Каждый модуль достаточно мал и его логику легко понять;
- Информация о каждой из таблиц ESTAB и RLTAB спрятана внутри одного соответствующего модуля;
- Только два модуля имеет доступ к формату объектных модулей произведенных компилятором и имеют сцепление по формату;
- Все остальные модули имеют сцепление по данным;
- Все операции чтения/записи для каждого файла сосредоточены в единственном модуле.

Напомним только, что реальный загрузчик иметь одну дополнительную деталь: он или сразу активизирует загруженную программу или вернет адрес ее точки входа обратившейся к нему программе.

ОКОНЧАТЕЛЬНАЯ ПРОВЕРКА

Ддя поиска ошибок или недостатков в проектировании приложения можно последовательно применить три метода:

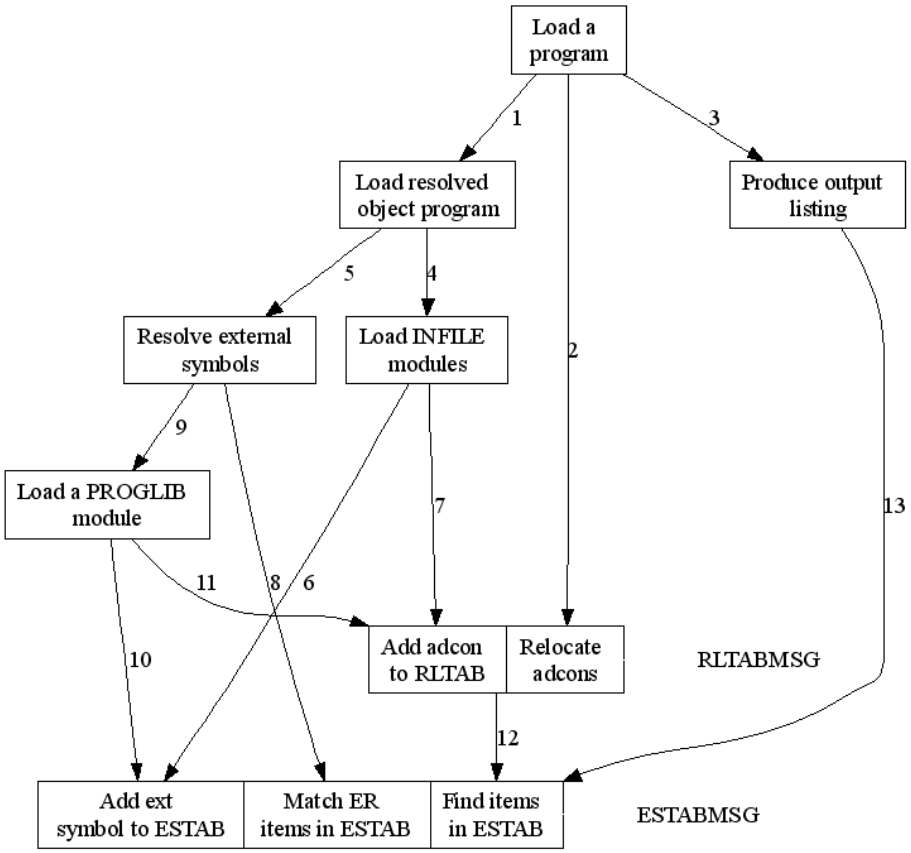
- проверку n-плюс-минус-один;
- статическую проверку;
- сквозной контроль.

Метод n-плюс-минус-один – это официальная проверка проектной документации разработчиками этапа n-1 (авторы архитектуры системы и внешних спецификаций), которые ищут ошибки понимания спецификаций и архитектуры системы (translation mistakes), и разработчиками уровня n+1 (производство внешнего проекта модуля), которые проверяют насколько осуществим и понятен проект, согласуется ли он с языком программирования и операционной системой, которые предполагается использовать.

Статической проверкой называется оценка проекта с точки зрения рассмотренных ранее в этой граве характеристик, выполняемых членами второй группы. Участников этой проверки должны интересовать следующие вопросы:

- все ли модули имеют информационную или функциональную прочность?
- Если нет, почему?
- Все ли модули имеют сцепление по данным?
- Все ли модули предсказуемые?
- Закончена ли декомпозиция? (например, возможно ли достаточно точно представить логику каждого модуля)
- Является доступ к данным у всех модулей минимальным?

Третий методом проверки является сквозной контроль, похожий на метод сквозного контроля, ранее рассматривавшемуся для предыдущих процессов проектирования. Надо спроектировать набор ситуаций для мысленного тестирования (paper rest cases) (например, схемы 2 и 3 можно использовать для создания набора ситуаций для загрузчика). Затем для каждого теста проследить за действиями системы шаг за шагом, двигаясь по структуре программы. При этом предполагается, что логика каждого отдельного модуля корректна (что каждый модуль выполняет свои функции правильно). Надо выявить такие недостатки в проекте, как отсутствующие функции, недостаточные интерфейсы (недостаточное количество параметров в интерфейсе) и не корректные результаты. Рассмотрите достаточное количество различных тестов чтобы каждый модуль был вызван по крайней мере один раз. Рассмотрите в качестве тестов и неправильные входные данные (например, может быть объектный модуль, у которого отсутствует запись ESD), а также тесты с 'граничными условиями' (например, объектный модуль без внешних ссылок или без адресных переменных).



Примечания

- LOAD-INFILE-MODULES использует функции операционной системы GET (из INFILE) GETMAIN (для распределения оперативной памяти).
- LOAD-A-PROGLIB-MODULE использует систем функции FIND и GET (из PROFLIB), и GETMAIN.
- PRODUCE-OUTPUT-LISTING использует системную функцию PUT (в OUTFILE).
- запись ESTAB содержит имя символа, тип (MD, EP, ER) и адрес распределенной памяти.
- запись RLTAB содержит номер соответствущей записи ESTAB и адрес адресной константы.

Рис. 8: Конечный результат

	in	Out
1		ESTAB,RLTAB,MSGLIST
2	ESTAB,RLTAB	EC
3	ESTAB,MSGLIST	EC
4		ESTAB,RLTAB,MSGLIST
5	ESTAB,RLTAB	ESTAB,RLTAB,MSGLIST
6	ESTAB,NAME,TYPE, ADDRES	ESTAB,ENTRY NO, EC
7	RLTAB,ENTRY NO, ADCON ADRESS	RLTAB,EC
8	ESTAB	ESTAB, DONEFLAG, NAME OF AN UNRESOLVERD ER
9	ESTAB, RLTAB, MODULE NAME	ESTAB, RLTAB, MSGLIST
10	ESTAB,NAME,TYPE, ADDRES	ESTAB,ENTRY NO, EC
11	RLTAB,ENTRY No, ADCON ADRESS	RLTAB,EC
12	ESTAB, ENTRY No	NAME, TYPE, ADDRESS, EC
13	ESTAB, ENTRY No	NAME, TYPE, ADDRESS, EC

Рис. 9: Список интерфейсов

Предметный указатель

ССЫЛКИ

[1] Технология программирования. Лекция 6. http://www.ergeal.ru/txt/archive/cs/tp/06.htm.

[2] А.Г. Пискунов, . Raise Specification Language: проектирование и декомпозиция потоков данных интерактивного приложения. http://i.com.ua/~agp1/ru/dataFlow.html.

[3] Г. Майерс. Надежность программного обеспечения, 1980.

[4] А.Г. Пискунов. Формализация парадигмы обьектно-ориентированного программирования: критика определения Гради Буча, 2007. http://i.com.ua/~agp1/ru/oopFormalizm.html.

[5] А.Г. Пискунов. The RAISE Method Group: АЛГЕБРАИЧЕСКОЕ ПРОЕКТИРОВАНИЕ КЛАССА, 2007. http://users.iptelecom.net.ua/~agp1/ru/ClassDesign.html.

[6] К.Дейт. Введение в системы баз данных, 1998. 6-е издание.

[7] D. E. Knuth. Structured Programming with GO TO statments, 1974. Computing Surveys, 6(4), 261-301.

[8] Holt R.C. Structure of Computer Programs: A Survey, 1975. Proceeding of the IEEE, 63(6), 879-893.

[9] J.B.Goodenough and D.T.Ross. The Effect of Software Structure on Reliability, Modifiability, Reusability, Efficiency: A Preliminary Analisys, 1975. Proceeding of the IEEE, 63(6), 879-893.

[10] Liskov B.H. A design Methodology for Reliable Software Systems. (Proceedings of the 1972 Fall Joint Computer, Montvale, N.J.: AFIPS Press, 1972, pp. 191 - 199).

[11] G. J. Myers. Software reability. principles and practices, 1976.

[12] Myers G.J. Reliable Softwart Throught Composite Design. (New York: Petrocelli, 1975).

[13] D. L. Parnas. On the criteria to be used in decomposing systems into modules, 1972.

[14] Spier M.J. A Critical Look at the State of our Science. (Operating Systems Review), 8(2), 9-15 (1974).

[15] Wulf W., Shaw M. Global Variable Considered Harmful. (SIGPLAN Notices), 8(2), 28-34 (1973).