Руслан Богатырев

Модульное программирование: Terra Incognita

Исторически сложилось так, что *модульное программирование* (modular programming) в своем классическом понимании оказалось для большинства программистов-практиков землей неизведанной, terra incognita. Из ведущих языков программирования (в порядке популярности — Си, Java, C++, Visual Basic, Delphi, C#, см. Р. Богатырев "Популярность языков программирования" — "Мир ПК", 01/2005), пожалуй, только Java и Delphi (со времен Turbo Pascal 4.x) имеют средства, в определенном смысле приближенные к классике. Концепция модуля, кстати, отсутствует в таких известных языках, как Smalltalk-80 и Eiffel.

Как же так произошло? Дело в том, что доминирующая сейчас ветвь языков Си-семейства не опиралась на эту концепцию, вытеснив ее в конце 1980-х годов другой, возведенной в разряд абсолюта — *объектно-ориентированным программированием* (ООП).

Если говорить коротко, то ООП опирается на идею совмещения в концепции *класса* (class) понятий *модуля* (module), *типа* (type) и *механизма расширения* (extension; наследования, обогащения). При этом в рамках ООП на основе классов непросто добиться полноценного воплощения возможностей триады *"модуль-тип-расширение"*, исповедуемой, в частности, языками Оберон-семейства. Приходится добавлять понятие пространства имен (namespace), охватывающее вместо одного несколько файлов, да еще и с поддержкой вложенности, вводить вместо понятного механизма экспорта-импорта запутанный набор средств по разграничению областей видимости (public, private, friend, protected и т.п.). Детальнее об этих проблемах см. в статье С.Свердлов "Язык программирования Си#: критическая оценка". Помимо *области видимости* в сфере языков программирования имеется и такое важное понятие, как *область существования*. А с ней чехарда не меньшая.

Появление понятия *компонента* (component), точнее, возрождение этого понятия на новом витке эволюции программирования в середине-конце 1990-х годов, остро поставило вопрос корректного отображения виртуально-мифического понимания классов на реально-осязаемое восприятие компонентов. Худо-бедно, в языках промышленного программирования, исповедующих диктатуру классов, задача решается, но весь вопрос в том, зачем это делать в подавляющем большинстве случаев таким неестественным образом? Само простое и логичное — отождествить физическое и отчуждаемое представление модулей с понятием компонента.

Модуль можно сравнить с устройством, имеющим средства сопряжения (подключения), через которые и происходит как управление самим устройством, так и обмен данными. Это своего рода черный ящик, для которого экспорт и импорт определяют его интерфейс и зависимость от других устройств.

Принципиальное отличие модулей от классов: *модули* — это уникальные экземпляры (второго такого в системе нет), не допускающие обобщения (generic-модули здесь не рассматриваем, они находятся вне подхода Вирта и в его языках не поддерживаются). Т.е. на их основе ничего порождать нельзя. Это важно.

Модули не только определяют четкие синтаксические границы кода (процедур) и данных, но также являются *единицами этапа компиляции* (unit of compilation) и *единицами этапа загрузки* (а также выполнения, замены — unit of replacement). Эти единицы можно редактировать, документировать, распространять и компилировать по отдельности. В модульном программировании связывание происходит на этапе загрузки (*динамическое связывание*, dynamic linking) и абсолютно невидимо пользователю.

Модуль — это контейнер для набора объектов, при этом он является средством абстрагирования кода и данных, ибо имеет две части: интерфейс и реализацию. Целостность данных и их защита обеспечивается в модулях за счет сокрытия информации (information hiding, или инкапсуляции) и физического вычленения данных и кода из создаваемой системы (программы). Причем при сочленении модулей гарантируется соблюдение всех требований безопасности типов (type safety).

Интерфейс в модульном программировании рассматривается как жесткая спецификация, контракт, который должен неукоснительно соблюдаться клиентами модуля (импортирующими его) и реализацией данного модуля (или несколькими альтернативными реализациями). Любое изменение контракта требует перекомпиляции/пересмотра всех зависимых модулей. Важно

отметить, что компилятор Оберона в системе ETH Oberon поддерживает режим расширения интерфейса, когда чистое расширение (добавление процедур) не влияет на ранее оттранслированные модули. В этом случае создается расширение бинарного представления интерфейса (символьного файла), что позволяет подстыковывать расширенную версию модуля без перекомпиляции ранее оттранслированных клиентских модулей.

Для модульного программирования характерно использование *раздельной компиляции* (separate compilation, см. R.Crelier "Separate Compilation and Module Extension" — ETH, 1994), когда компиляция интерфейса и реализации модуля делается отдельно от реализации других модулей, но с обязательным участием интерфейсов всех прямо и косвенно импортируемых модулей. Если нет возможности отделить интерфейс в текстовом или бинарном виде от реализации модуля, то такую систему программирования нельзя считать системой раздельной компиляции, ибо нарушается главный принцип — компиляции модуля при отсутствии в пределах досягаемости компилятора реализаций импортируемых им модулей.

Все ошибки несостыковки обнаруживаются на этапе трансляции (причем даже до реализации других модулей, ведь нужны только контракты-интерфейсы). Более примитивная и распространенная схема *независимой компиляции* (independent compilation) не использует эту информацию и вынуждена заниматься выявлением расхождений в сопряжении файлов лишь на этапе компоновки (linking). Высокая гибкость в модульном программировании достигается за счет совмещения фазы компоновки и загрузки. А в некоторых случаях (реализация Оберона на уровне подхода Oberon Module Interchange, предложенного Микаэлем Францем в 1994 г., см. "Динамическая кодогенерация: ключ к разработке переносимого ПО") совмещается на этом этапе (перед компоновкой) и фаза генерирования машинного кода для конкретной целевой архитектуры. Т.е. *безо всякой виртуальной машины* достигается эффективное выполнение модулей на любой операционной платформе.

В отношении области существования и области видимости. Процедура (а также объявленные в ней данные) существует ровно то время, пока не произойдет возврат из нее в точку ее вызова, и имеет абсолютно прозрачную по импорту и непрозрачную по экспорту область видимости (одностороннюю). Модуль (и его данные) существует все то время, пока загружен в память. Область его видимости регулируется средствами явного экспорта-импорта идентификаторов (импорта модулей; экспорта процедур, типов, констант, переменных, отдельных полей составных переменных).

Инициализация всех модулей, затрагиваемых данной программой, производится в порядке, обратном их глубине зависимости по импорту. Другими словами, перед началом работы программы производится т.н. топологическая сортировка, которая позволяет выявить последовательность проведения инициализации в направлении от абсолютно независимых по импорту модулей к максимально зависимым (а самым зависимым является, естественно, программный модуль — MODULE, т.е. программа).

Модуль на уровне своего содержимого устанавливает важный принцип *No Paranoia Rule* (никакой паранойи, см. Clemens Szyperski "Import is Not Inheritance. Why We Need Both: Modules and Classes" — ETH, 1992). Иными словами, внутри одного модуля можно размещать несколько классов, при этом общение между их полями и методами абсолютно прозрачно внутри, тогда как снаружи может регулироваться средствами избирательного экспорта (т.е. спецификаторами экспорта). Схема инкапсуляции в одном модуле строго одного класса практически соответствует общепринятой модели ООП. Иными словами, программист волен выбирать степень концентрации нескольких классов в одном модуле. Это важное преимущество Оберона (1988) по сравнению с другими языками ООП.

В языке Modula-2 (1979) для получения возможностей тонкого управления экспортом-импортом на локальном уровне (в рамках программных и исполнительных модулей) Вирт ввел понятие локального модуля. Однако практика показала, что это излишний и запутанный механизм, который в Обероне был уже исключен.

Если продолжать разговор о Modula-2 (как прямом предшественнике Оберона), то здесь есть тонкий момент. Это разная трактовка терминов *definition* (описание) и *declaration* (объявление, определение). Аналогично и для глаголов: define — описывать, declare — объявлять, определять. Definition подразумевает экспорт, соответственно, и информацию там требуется приводить ровно в том объеме, которая необходима для реализации принципов сокрытия информации. Описание может доопределяться, а объявление (определение) носит законченный характер. Но объявление можно расширять (или обогащать, удачный термин проф. В.Ш.Кауфмана из МГУ), напр., как это делается в языках Оберон-семейства. Если следовать такой терминологии, то в Обероне, в отличие от Modula-2, нет надобности делать описания. Достаточно использовать только объявления, в которых помечать звездочками (спецификаторами экспорта) те элементы, которые и должны быть видны снаружи, т.е. должны сформировать описание.

В языке Modula-2 различие между описанием и объявлением (определением) прослеживается на уровне DEFINITION MODULE (описательный модуль) и IMPLEMENTATION MODULE (исполнительный модуль). В описательном модуле можно не только давать *описание* (скрытых типов, процедур), но и приводить *объявление* (типов, констант, переменных, но не процедур!). К одному описанию (definition) можно готовить несколько разных реализаций (implementation) и подменять их (на этапе компоновки) в зависимости от требований (точности вычислений, быстродействия, особенностей алгоритма и т.п.)

В Modula-2 допускался циклический импорт на уровне DEFINITION (не страшно, ибо это просто ссылающиеся друг на друга описания), но запрещался на уровне IMPLEMENTATION. В Обероне эта проблема отпала сама собой. Здесь из-за текстуального слияния интерфейса и реализации, а также для предотвращения ненужных проблем циклический импорт запрещен.

В Обероне сохранилось деление на описательный (интерфейсный) и исполнительный модули с той принципиальной разницей, что описательный модуль готовится теперь уже не проектировщиком (программистом), как в Modula-2, а автоматически формируется компилятором из исполнительного модуля на основе спецификаторов экспорта (звездочек у каждого экспортируемого объекта, включая отдельные поля записей). Относительным неудобством стала необходимость для получения спецификации (интерфейса) фиктивной (вырожденной) реализации модуля. Зато во всем остальном были весьма элегантно решены проблемы модульного программирования.

Очень важная особенность Оберона — использование исключительно квалифицирующего импорта (тогда как в Modula-2 допускался и неквалифицирующий), т.е. помимо указания модуля в списке импорта данного модуля в самом тексте импортирующего модуля каждый импортируемый объект указывается с явным префиксом имени модуля, например, IMPORT Math; ... Math.sin(x), где Math — это модуль, реализующий базовые математические функции. В Обероне был введен механизм синонимизации имен, когда на уровне списка импорта можно задать соответствие внешнего имени модуля его локальному синониму (что удобно для исключения коллизиций и для явной коммутации модулей).

В Обероне появилось еще одно интересное решение, связанное с модульным программированием. Понятие программы исчезло. Вместо этого Вирт ввел концепцию *команд* (command). Команды — это экспортируемые процедуры без параметров, определяющие точки входа (вызова) программы. Другими словами, программа превратилась в модуль, экспортирующий по сути сервисы. В ОС Oberon команды становятся полноправными командами операционной системы, которые можно напрямую запускать, связывать в последовательность обработки и т.п. Любые модули (библиотечные и программные — разницы между ними нет) можно динамически загружать и выгружать (без перекомпиляции).

Чтобы детальнее разобраться в природе модульного программирования, давайте обратимся к истории.

До начала 1970-х годов программы создавались в виде монолитных блоков, либо делались из независимых частей, сопряжение которых было достаточно примитивным — на уровне вызовов подпрограмм (процедур). Отсюда и пошли два известных понятия — *цельная компиляция* (whole compilation) и *независимая компиляция* (independent compilation). Первый случай простой и пояснений наверняка не требует (транслируется вся программа целиком). Во втором каждый блок (файл) транслируется отдельно, фактически без наличия информации о точках сопряжения. Все проблемы увязки возлагались на *компоновщик* (linker). Именно он состыковывал оттранслированные части, соединял программу с библиотеками, которые та использовала.

Лобовое решение этой проблемы нашло свое отражение в языке Си в виде знаменитых директив препроцессора (#include). Здесь проблема не решалась, а запрятывалась вглубь. Вместо строгого контроля и четкого взаимодействия механизмов экспорта-импорта, выделения областей видимости и существования предлагалось использовать сокращенную запись операций скрытого расширения исходного текста на этапе, непосредственно предшествующем компиляции (препроцессинга).

Не буду вдаваться в подробности относительно проблем #include, другими языками Си-семейства, особенно C++. Об этом написано предостаточно. См. напр., Питер Мойлан "Аргументы унаследованных против Си" (1993), M. Sakkinen "The Darker Side of C++", I.Joyner "A Critique of C++ and Programming and Language Trends of the 1990s" (1996).

В каноническом Паскале (в трактовке Вирта) данный вопрос вообще никак не решался, и это было безусловно ахиллесовой пятой языка. Но в конце 1970-х годов сразу три языка включили в свой арсенал эффективный механизм модуля. Это сделали CLU (1973, Барбара Лисков, Массачусетский технологический институт, США) — понятие *кластера* (cluster), Modula-2 (1979, Никлаус Вирт, ETH Zurich) — понятие *модуля* (module) и Ada (Джин Ихбиа и др., 1980, Министерство обороны США) — понятие пакета (package).

Однако, пожалуй, первым наиболее явно это сделал язык Mesa (1973, Джеймс Митчелл и др., Xerox PARC), — язык, положенный Виртом в основу Modula (1976), а потом и Modula-2 (1979) после года работы Вирта (1976—1977) в стенах Xerox PARC.

Всей этой четверке языков предшествовали научные исследования, отчеты о которых выходили в очень компактный промежуток времени (1971-1974).

Сначала в апреле 1971 г. в Communications of the ACM появилась статья Никлауса Вирта из Высшей политехнической школы ETH (Швейцария) "Разработка программ методом пошагового уточнения" (Niklaus Wirth "Program Development by Stepwise Refinement". В ней не использовалось слово "модуль", но при этом на примере классической задачи по расстановке 8 ферзей на шахматной доске были сформулированы подходы к декомпозиции (разбиению) монолитной программы на набор задач (действий, инструкций), каждая из которых при очередном шаге уточнения получает все более высокую степень конкретизации, при этом затрагивая конкретизацию и данных. Это устанавливало иерархию абстракций на уровне операций и данных.

В декабре 1972 г. в Communications of the ACM была опубликована статья Дэвида Парнаса из университета Карнеги-Меллон "О критериях по декомпозиции систем на модули" (David Parnas "On the Criteria To Be Used in Decomposing Systems into Modules").

Помимо собственно декомпозиции системы важным критерием модуляризации Парнас назвал сокрытие информации (information hiding, "скрывать решение от других"). За счет использования средств внешних модулей (слова "использует", "зависит от") формируется иерархическая структура, которая задает отношение частичного порядка, подразумевающего проведение топологической сортировки.

В работе "A History of CLU" Барбара Лисков (Barbara Liskov) из Массачусетского технологического института вспоминает: "В 1972 г. я предложила идею *разделов* (partitions). Система делится на иерархию разделов, каждый из которых представляет один уровень абстракции и состоит из одной или нескольких функций, оперирующих общими ресурсами… Связь на уровне данных между разделами ограничена использованием явных аргументов, передаваемых из функций одного раздела во (внешние) функции другого раздела. Неявное взаимодействие с общими данными осуществляется только среди функций, лежащих внутри соответствующего раздела…"

Далее она продолжает: "Это привело меня к пониманию связывания модулей с типами данных и к идее *абстрактных типов*, имеющих инкапсулированное представление и операции, которые могут быть использованы для доступа и манипулирования объектами… Я называла типы *абстрактными*, поскольку они не предоставляются напрямую языком программирования, а вместо этого должны реализовываться пользователем. Абстрактный тип является абстрактным точно в таком же смысле, как то, что процедура является абстрактной операцией".

В начале 1973 г. Барбара Лисков на фоне разочарования работами по методологии программирования и зачатками модульного программирования в развитие идеи разделов создала новую концепцию — кластер (cluster), что и привело к образованию нового языка — CLU.

Концепция модуля как основы сокрытия информации (information hiding) тесно переплелась с концепцией абстрактных типов данных (ADT, abstract data types), поскольку введение различных уровней абстракции и обеспечивалось средствами контроля областей видимости со стороны модуля.

В октябре 1974 г. вышла известная статья Тони Хоара "Monitor: An Operating System Structuring Concept". Годом ранее в работе "Operating System Principles" Пер Бринч Хансен ввел аналогичное понятие *"shared"*. Впоследствии этой идее дали название *мониторы Хансена-Хоара*. Это особая форма модуля, в который заключены процедуры и соответствующие структуры данных, при этом доступ к модулю (его процедурам) для внешних процессов является взаимоисключающим.

Итак, к середине 1970-х годов понятие модуля стало обретать все более ясные очертания, при этом сформировались две специфики его применения — мультипрограммирование (монитор) и абстрактные типы данных (кластер). Все это нашло отражение в языке Modula-2, где роль мониторов выполняли модули с приоритетами (в них реализовывались драйверы устройств), а абстрактные типы данных воплощались в понятии скрытых типов (opaque type). Последние на уровне описательного модуля (DEFINITION MODULE) выглядели простым названием без объявления структуры, а на уровне исполнительного модуля (IMPLEMENTATION MODULE) реализовывались, как правило, указателями на комбинированный тип (RECORD). В Обероне мониторы, как и другие средства мультипрограммирования Modula-2, вынесены за пределы языка, а абстрактные типы данных реализуются средствами частичного (избирательного) экспорта. При этом за счет механизма расширения типа полностью реализуют привычную парадигму ООП, но это уже тема другой статьи.