Руслан Богатырев

Особенности поддержки концепции модуля в языках Delphi (Object Pascal), Modula-2 и Оберон

При сравнении языков Delphi (Object Pascal) и Оберон добавлены комментарии в отношении языка Modula-2, поскольку это прямой предшественник языка Оберон и его решения иллюстрируют эволюцию восприятия модульного программирования Никлаусом Виртом.

В качестве описаний языков использовались

- [1] Object Pascal Language Guide [Delphi 6] (2001)
- [2] Borland Delphi 8 for .NET (2004)
- [3] Delphi Language Guide. Delphi for Microsoft Win32 and .NET (2004)
- [4] Niklaus Wirth. The Programming Language Modula-2. 4th Edition (1988)
- [5] Niklaus Wirth. The Programming Language Oberon (1990)

В качестве эталонных систем программирования для сравнения использовались:

1. Borland Delphi 6 / Borland Delphi 2005
2. ETH Oberon (до 2000 г. носила название Oberon System 3)

Следует заметить, что в данном сравнении не ставится целью определить победителей и проигравших. Задача в том, чтобы попытаться раскрыть Delphi-программистам (и тем, кто программирует на других языках, но знаком с Delphi) некоторые нюансы поддержки модульного программирования в тех языках, чье развитие шло параллельным (и не известным большинству из них) курсом (речь о Modula-2 и Обероне).

Виды модулей

В языке Delphi понятие модуля реализуется конструкцией **unit**. В дальнейшем будем называть ее Delphi-модулем. Каждый Delphi-модуль состоит из четырех секций, следующих друг за другом в жестком порядке в одном **unit**-файле — интерфейса (**interface**), реализации (**implementation**), инициализации (**initialization**) и финализации (**finalization**).

В языке Оберон существует только один вид модулей, определяемых ключевым словом **MODULE** (на самом деле, до 1990 г. язык Оберон поддерживал деление модулей по принципу Modula-2, см. ниже). Программа также является модулем. Она ничем не отличается от других модулей, а работа с ней осуществляется посредством команд — экспортируемых процедур без параметров, которые формируют точки входа (сервисы) этой программы (более подробно см. Модульное программирование: Terra Incognita). В языке Modula-2 используются библиотечные модули (состоят из двух частей — **DEFINITION MODULE** и **IMPLEMENTATION MODULE**) и программные модули (программа, **MODULE**). В языке Delphi есть программа (**program**), библиотеки (**library**) и **unit**-модули. Для них действуют несколько различные правила.

Особенности экспорта

Интерфейс в Delphi служит спецификацией экспорта. Здесь описываются публично доступные (за пределами данного Delphi-модуля) константы, типы, переменные, процедуры и функции, принадлежащие данному Delphi-модулю. Важная особенность: в интерфейсе описываются (декларируются) все составляющие класса (class members), включая поля (fields), методы (methods, точнее, процедуры и функции) и свойства (properties) класса, распределенные по пяти поддерживаемым категориям видимости (**private**, **protected**, **public**, **published** и **automated**). В языке Modula-2 интерфейс заключен в выделенный описательный модуль (**DEFINITION MODULE**). Иными словами, нарушается основополагающий принцип инкапсуляции — программист, использующий модуль, видит в интерфейсе детали реализации, которые ему не могут быть доступны.

В языке Оберон интерфейс (в текстовом — для пользователя — и бинарном — для компилятора — видах) формируется на основе специальных значков экспорта (звездочек),

выставляемых сразу за экспортируемыми идентификаторами. Такие значки называют спецификаторами экспорта. В Обероне соответственно интерфейс содержит только ту информацию, которая является публичной (экспортируемой).

В языке Delphi в разделе реализации (**implementation**) допускается опускать списки параметров у экспортируемых процедур и функций (считая, что точный порядок и способ передачи заданы в интерфейсе). Нельзя сказать, что это является источником многочисленных ошибок, но вводит ненужную путаницу (в интерфейсе одно, в реализации — другое).

Коллизии неизбежны, если, например, сигнатура (заголовок) процедуры присутствует в интерфейсе и реализации, которые текстуально разделены и правятся программистом. Программисты допускают ошибки (по невнимательности, из-за эволюционирования проекта и т.п.). Компилятор на них указывает. Их можно исправить, но они возникают. В Обероне такие ошибки не могут возникнуть в принципе. Выходить из ситуации можно по-разному. Вопрос: зачем ее создавать вообще?

В языке Modula-2 требуется соблюдать четкое соответствие между заголовками процедур (процедур-функций) в описательном (**DEFINITION MODULE**) и исполнительном (**IMPLEMENTATION MODULE**) модулях. В языке Оберон такой проблемы нет, поскольку заголовок процедуры записывается ровно один раз (и помечается звездочкой в случае экспорта).

Импорт и инициализация модулей

В Delphi как интерфейс, так и реализация имеют собственные списки импорта (ключевое слово **uses**). В них перечисляются импортируемые Delphi-модули. Будем также называть эти списки **uses**-списками. Конструкция **uses**-импорта в Delphi использует возможность конкретизации физического размещения импортируемого **unit**-модуля, т.е. использование **uses** A **in** '../b.pas' дает команду компилятору искать соответствующий **unit**-модуль в файле по указанному пути. Это справедливо для программного модуля (**program**) и для библиотек (**library**), но не разрешено для **unit**-модулей. Подобный подход вносит в исходный текст жесткую зависимость от файловой системы.

Поскольку в Delphi **interface**-импорт (т.е. импорт в **interface**-части) автоматически расширяет область видимости идентификаторов для **implementation**-части, то нет необходимости дважды импортировать один и тот же модуль (сначала в **interface**-импорте, затем в **implementation**-импорте). В описании языка [1] рекомендуется во избежание проблем с циклическим импортом (подробнее см. об этом ниже) по возможности воздерживаться от **interface**-импорта и ограничиваться **implementation**-импортом. Но бывают случаи, когда **interface**-импорт необходим (например, при импорте константы, которая участвует в описании нового типа данных на уровне **interface**-части). Таким образом, возникает еще одна неопределенность. В языке Оберон этой проблемы в принципе быть не может.

Секции инициализации и финализации в языке Delphi являются по сути анонимными процедурами без параметров, которые вызываются соответственно при старте программы (после загрузки Delphi-модулей) и при завершении программы (перед выгрузкой Delphi-модулей). В Modula-2 и Обероне секция инициализации выделяется парой **BEGIN**-**END** имя_модуля. Финализации нет (за исключением ISO-стандарта языка Modula-2 и языка Компонентный Паскаль, который является "внуком" Оберона).

Важный момент: порядок инициализации импортируемых Delphi-модулей (для данного модуля) определяется порядком их перечисления в **uses**-списке импорта (данного Delphi-модуля). Порядок финализации — строго обратный порядку инициализации. Вопрос: зачем надо было вводить жесткий ручной порядок инициализации вместо автоматического, осуществляемого с помощью топологической сортировки (как в Modula-2 и Обероне)? Ведь топологическая сортировка (обеспечивающая в рамках отношения частичного порядка построение списка от наименее зависимого к наиболее зависимому модулю) гарантирует формирование четкой цепочки зависимости по импорту. Все решает за человека математика.

В случае же Delphi надо в голове держать нюансы того, в каком таком месте стоит импортируемый модуль в списке импорта. Сейчас мы поставим его вперед — будет один порядок. Затем поставим назад — другой. Если учесть, что при инициализации модулей могут активно потребляться (распределяться, резервироваться и т.п.) ресурсы компьютера, то проблема станет еще более рельефной.

Другой вопрос, возникающий при чтении описания [1] языка Delphi: какой из двух **uses**-списков импорта (в интерфейсе или в реализации) влияет на порядок инициализации импортируемых

модулей (что будет, если они перечислены по-разному)? В описании [3] для Delphi for .NET внесено уточнение: речь идет именно об **interface**-списках. Но запутанность ситуации такое уточнение все же не устранило.

Проблема определения этого усугубляется тем, что Delphi поддерживает и третью форму существования импорта — неявный импорт на уровне языка (но явный на уровне файла проекта, т.е. на уровне системы программирования).

Существует, кстати, еще один неявный импорт в Delphi — импорт псевдомодуля System (содержащего операции обработки строк, ввода-вывода, работы с динамической памятью и т.д.). Он производится автоматически для каждого Delphi-модуля. Сравните это с подходом Modula-2 и Оберона, где псевдомодуль SYSTEM заключает в себе низкоуровневые средства программирования, и его явный импорт является индикацией зависимости модуля от аппаратной/операционной платформы (т.е. влияет на переносимость).

Раздельная компиляция

Язык Delphi поддерживает ограниченную форму раздельной компиляции: раздельную в смысле разделения при компиляции исходного текста и бинарного, оттранслированного файла для импортируемых модулей, но не раздельную в смысле разделения бинарного вида интерфейса и бинарного вида реализации модуля. Это означает, что для компиляции Delphi-модуля, импортирующего другие, достаточно иметь все их интерфейсы (в бинарном виде) без наличия в области досягаемости компилятора исходных текстов **unit**-модулей. В Delphi (Win32) бинарные интерфейсы хранятся в .dcu-файлах (в Linux — в .dpu-файлах; в .NET — в .dcuil-файлах) по аналогии с тем, как в Modula-2 и Обероне бинарные интерфейсы хранятся в .sym-файлах (бинарное представление интерфейсов — это таблицы символов). Но принципиальная разница в том, что в .dcu-файлах хранится и реализация, т.е. нет возможности на уровне средств системы программирования подстыковывать разные реализации, воплощающие один и тот же интерфейс, а в Modula-2 и Обероне обеспечивается возможность поддержки полноценной раздельной компиляции. Более того, в .sym-файлах при компиляции записывается уникальный код (timestamp), отражающий момент последней компиляции (существуют реализации, использующие вместо этого уникальный "отпечаток пальца" каждого экспортируемого элемента модуля). В .obj-файле, являющемся результатом раздельной трансляции исполнительных модулей (реализации), для каждого импортируемого модуля помимо его имени заносится и этот уникальный код.

Таким образом, в Обероне гарантируется высокая надежность и целостность системы, что особенно важно при динамической замене/перезагрузке модулей.

Нюанс использования раздельной компиляции состоит в том, что при командной разработке нередко требуется жестко зафиксировать текстовый и бинарный интерфейс, оставляя при этом возможность другим разработчикам вносить исправления в реализацию соответствующего импортируемого модуля без необходимости полной перекомпиляции и пересборки проекта. В случае совмещенного бинарного хранения интерфейса и реализации (как в Delphi) это невозможно. В то время как в Обероне можно потом загрузить (без перекомпиляции и перекомпоновки) реализацию импортируемого модуля, если только он соответствует интерфейсу.

Delphi не предоставляет и выделенного отчуждаемого текстового представления интерфейсов. В Modula-2 это отдельный .def-файл (описательный **DEFINITION**-модуль), а в Обероне — .def-файл, формируемый компилятором (в ETH Oberon — утилитой Watson) из исходного .mod-файла на основе спецификаторов экспорта (звездочек). Надо заметить, что в некоторых реализациях Modula-2 и Оберона выделенный .sym-файл не используют, применяя ограниченную форму раздельной компиляции.

Конфликт имен

Язык Delphi поддерживает два вида импортируемых идентификаторов: неквалифицированные и квалифицированные. В исходном тексте Delphi-модуля импортируемые идентификаторы обычно используются как есть (неквалифицирующий импорт), без префиксирования именем соответствующего модуля. Могут возникать ситуации, когда идентификатор с одним и тем же именем (напр., константа maxndx) описан в двух разных Delphi-модулях, причем оба импортируются данным модулем. Как быть? В этом случае применяется "правило последнего", т.е. без всякого сообщения об ошибке коллизия считается допустимой и применяется

идентификатор из того модуля, который в **uses**-списке импорта стоит ближе к концу (т.е. отменяет совпадение предшествующего).

Если требуется явно указать конкретный идентификатор, то Delphi предоставляет возможность квалифицированных идентификаторов (напр., MyModule2.maxndx). В языке Modula-2 также поддерживаются оба вида импортируемых идентификаторов, но без права разрешения коллизии по умолчанию (это считается ошибкой, обнаруживаемой компилятором). В языке Оберон допустимы только квалифицированные идентификаторы, что снимает подобные проблемы. Более того, для удобства использования Оберон поддерживает синонимы модулей. Другими словами, в списке импорта можно вводить соответствие внешнего имени (импортируемого модуля) его внутреннему имени в данном модуле (синоним, псевдоним). Это позволяет снять зависимость по разработке модулей разными группами, которые не координированы по формированию имен.

Если внутри модуля используется внешний неквалифицированный идентификатор, то в тексте (особенно в чужом) Delphi-программисту вряд ли удастся быстро определить, импортируется ли он извне или является локальным идентификатором одной из сущностей данного модуля (константы, переменной, типа, процедуры). Что лучше — жесткое правило всегда снабжать импорт "акцизной маркой" (то есть префиксировать именем модуля, как в Обероне) или не делать (обычно, ибо лень) никаких различий (как в Delphi и Modula-2)? Оберон дает ясный ответ на этот вопрос.

Циклический импорт

Язык Delphi допускает прямую или опосредованную циклическую зависимость модулей. Циклический импорт на самом деле таковым не является, поскольку, как известно, в Delphi существует **interface**-импорт и **implementation**-импорт, при этом циклическая зависимость для **interface**-импорта запрещена. Она разрешена только для **implementation**-импорта. В языке Modula-2 существует подобное правило для **DEFINITION**-импорта и **IMPLEMENTATION**-импорта. В Обероне из-за наличия одного файла (**MODULE**) со спецификаторами экспорта вообще нет деления импорта на **DEFINITION**-импорт и **IMPLEMENTATION**-импорт, поэтому циклический импорт запрещен. Кроме того, многолетняя практика использования Modula-2 показала нецелесообразность и потенциальную опасность циклического импорта. В случае необходимости всегда можно реализовать потребности в общих импортируемых описаниях через дополнительный модуль.

Расширение модулей

В языке Оберон поддерживаются средства расширения экспорта, удовлетворяющие требованиям раздельной компиляции и строгой типизации. Это означает, что после того, как сформирован и оттранслирован модуль, он может быть расширен даже в случае использования его основного интерфейса другими модулями. Данный механизм реализуется через скрытый расширенный модуль (на уровне .sym-файлов). Расширение должно быть бесшовным (например, в виде добавления процедур). Более подробно см. "О расширяющем программировании в языке Оберон". Языки Delphi и Modula-2 подобной возможности не имеют.

Пространства имен

Пространства имен (namespaces) при наличии модулей являются избыточным понятием, которое, тем не менее, введено в Delphi for .NET (Delphi 2005) как дань привязки к платформе Microsoft Common Language Runtime (CLR). Пространства имен, выступая в роли абстрактных контейнеров, добавляют правила регулирования областями видимости, задают порядок поиска идентификаторов при компиляции и поддерживают иерархию пространств. В одно пространство имен можно включать несколько Delphi-модулей.

Пространства имен помогают разрешить коллизии при совпадении имен разных модулей (напомним, что в Обероне это решается простым механизмом локальной синонимизации), заключая их в свою оболочку. В CLR для идентификации классов и типов используют квалифицированные идентификаторы, где сначала указывается имя сборки (assembly), затем пространство имен, которое содержит тип. Если для языков, не поддерживающих модульное программирование, пространство имен позволяет как-то обеспечить имитацию понятия модуля, то для того же Delphi такая избыточность (модуль + пространство имен) создает серьезную и

ненужную путаницу. Остается добавить, что именование пространств имен не различает больших и маленьких букв (смешение регистров).

12 важных отличий Оберона от Delphi. Модульное программирование

1. Единый вид **Оберон**-модулей против нескольких видов **Delphi**-модулей (программа, библиотека, **unit**).
2. Один вид импорта в **Обероне** против двух в **Delphi** (**interface**-импорт и **implementation**-импорт).
3. Использование только квалифицирующего импорта (префиксирование идентификатора именем модуля) в **Обероне** против коллизии имен в **Delphi**.
4. Механизм локальной синонимизации имен модулей в **Обероне** против вложенных пространств имен в **Delphi 2005**.
5. Вынесение в интерфейс только экспортируемых идентификаторов в **Обероне** против введения дополнительных правил ограничения экспорта в **Delphi** (классы).
6. Автоматическое отсутствие рассогласований между интерфейсом и реализацией в **Обероне** против коллизий в **Delphi**.
7. Запрет циклического импорта в **Обероне** против его поддержки в **Delphi**.
8. Инвариантность модулей относительно файловой системы в **Обероне** против привязки модулей к файловой системе в **Delphi** (на уровне импорта).
9. Автоматическое установление порядка инициализация модулей в **Обероне** против ручного порядка в **Delphi**.
10. Наличие автоматически отчуждаемой текстовой части интерфейса в **Обероне** против жесткого текстуального совмещения интерфейса и реализации в **Delphi**.
11. Расширенная поддержка раздельной компиляции в **Обероне** (разделение интерфейса и реализации как на уровне текстового, так и на уровне бинарного представления) против частичной поддержки в **Delphi**.
12. Возможность расширения интерфейса без перекомпиляции клиентских модулей в **Обероне** (расширяющее модульное программирование) против нерасширяющего модульного программирования в **Delphi**.

Наконец, главное отличие: **Оберон**-модуль является унифицированной единицей компиляции, компоновки, загрузки и замены (на лету), т.е. выполняет роль полноправного программного компонента, предоставляющего внешние сервисы. **Delphi**-модуль такими возможностями не обладает.

В системе Oberon System 3 (ETH Oberon) компоненты (gadgets) являются долговременными объектами (persistent objects). Они бывают как визуальными (видимые на экране), так и не визуальными (работающие в фоновом режиме). Подобно тому, как язык Оберон не различает программу и библиотечный модуль (давая полноправную возможность расширять ОС своими командами), система Oberon не делает различий между приложениями и документами. Файл, подобно документу, состоит из долговременных объектов. Открытие документа равносильно запуску приложения. Таким образом, документ и приложение состоят из одних и тех же строительных блоков — компонентов.