BLACKBOX.METASYSTEMS.RU «BLACKBOX ПО-РУССКИ!»
mailto:bb16@metasystems.ru
10 января 2007 г.

Некоторые идеи архитектуры Оберон-систем

И. Е. Ермаков

Введение

Мотивом к написанию данной статьи стали долгие дискуссии на форумах и необходимость из раза в раз повторять одно и то же по мере включения в разговор новых людей, незнакомых с Оберон-технологиями. В статьи кратко изложены некоторые эффективные архитектурные приемы, которые поощряются языками Оберон-семейства и, в частности, Component Pascal, и используются в операционных системах и средах этого семейства. Ключевой идеей для всех этих приемов является идея компонентного программирования и динамически раширяемых программных систем.

Рассмотрим следующие идеи:

1. особенности ООП в Оберонах;
2. родовые шины сообщений (generic message buses);
3. инсталлируемые каталоги объектов (directories);
4. выгрузка и перезагрузка модулей в компонентных системах.

1 Особенности ООП в Оберонах

Этот момент является важным для понимания всех последующих приемов. Он неоднократно затрагивался в других публикациях по теме Оберонов.

В Оберонах принята концепция «класс = тип данных». Во-первых, в язык не вводятся дополнительные концепции класса и объекта, для структурного языка с развитой системой типизации это просто излишне. Достаточно ввести механизмы расширения структур данных и связывания процедур с типами, которое дает полиморфизм их вызова и удобную форму записи `object.Procedure`. Указатель на объект всегда записывается как явный параметр связанной процедуры, что повышает ясность кода и не порождает сложностей с пересечением в методах нескольких пространств имен. Из других языков аналогичный подход был использован в Ada-95.

Во-вторых, ООП в Оберонах не несет архитектурной нагрузки. В суррогатных индустриальных языках на класс возлагаются две принципиально различные функции — типа данных и модуля. В модульных языках Паскаль-семейства модуль является отдельной синтаксической конструкцией — единицей исходного кода, инкапсуляции, компиляции, а для Оберонов к тому же — распространения, развертывания и загрузки в память. Перенос инкапсуляции на уровень модулей является важным решением, это позволяет процедурам и типам одного модуля свободно взаимодействовать между собой, без необходимости обходить инкапсуляцию и без накладных расходов, обычно

связанных с этим. Этот момент является одним из факторов эффективности Оберона для системного программирования.

Сочетание модульности и «компактного ООП» позволяет синтаксически развести абстракции архитектуры и абстракции проблемной области. Несколько упрощая, можно сказать, что модули являются элементами архитектуры, типы данных, в том числе объектно-ориентированные, являются элементами модели проблемной области.

Остановимся также на анализе сущности ООП в программных системах. Часто встречающееся «банальное» понимание ООП зациклено на терминах «класс» и «объект» и трактует их роль весьма поверхностно, исключительно с точки зрения удоства использования этих синтаксических конструкций прикладным программистом в его задачах. В глубоком понимании ООП имеет два смысла: прикладной и технический.

Прикладное ООП предполагает на этапе анализа предметной области выделение обособленных сущностей, обладающих состоянием и поведением — объектов, — и классов объектов, имеющих одинаковую структуру и поведение, затем выделение отношений «род-вид» и построение иерархий наследования для классов.

Техническое ООП предполагает: 1) возможность написания кода, работающего с типами данных, неизвестными во время его написания, что обеспечивается синтаксическим механизмом расширения структур данных и совместимостью типов по правилу «если A' расширяет A, то A' может использоваться всюду, где A»; 2) возможность связывания поведения с объектом данных, при этом каждый экземпляр данных сам «знает» свое поведение. То есть, выполняется позднее связывание точки вызова некоторой процедуры с реальным кодом, который ее будет выполнять. Код, работающий с объектно-ориентированными типами данных, имеет в себе некоторую недетерминированность, которая разрешается только в момент выполнения кода в зависимости от конкретного экземпляра данных, с которым работает код. Такая позднее связывание еще называют **виртуализацией** кода.

При таком понимании оказывается, что объектная ориентированность «не сошлась клином» на наличии в языке классов и объектов. Более того, введение этих понятий в языки, в которых техническая сторона ООП уже поддерживается другими средствами, является бессмысленным решением. Ярким примером стали объектно-ориентированные модицикации LISP. Проектируя Оберон, Вирт прекрасно понимал техническую сущность ООП и обошелся минимальными расширениями структурного модульного языка, чтобы сделать его объектно-ориентированным.

Пример 1. ООП в Oberon[1].

```
TYPE
    Figure = RECORD
        color: INTEGER
        Square: PROCEDURE (VAR f: Figure): REAL
    END;

    Rectangle = RECORD (Figure)
        x0, y0, x1, y1: REAL
    END;
```

[1] См. N. Wirth. The Programming Language Oberon. Revision 01.10.1990. (Здесь и далее сноски редактора — Б. В. Рюмшина.)

Для связывания процедур с объектами использовались поля процедурного типа, то есть, никакой специальной синтаксической поддержки методов не вводилось. Первым достоинством такого подхода является более сильная виртуализация — связывание вызова может варьироваться не только в зависимости от типа, но и от конкретного экземпляра объекта. Вторым достоинством является возможноть динамического связывания объекта с методом (такие «инсталлируемые методы» — процедурные поля — применяются и в других языках, например, в Delphi VCL, для динамического связывания объектов с обработчиками событий). Недостатками является необходимость при создании экземпляра объекта выполнять так называемую **инсталляцию методов** — инициализацию процедурных полей, а также громоздкая форма вызова `object.Proc(object, ,,,)`. Тем не менее, такое решение позволило успешно решать системные задачи и реализовать передовые для своего времени операционные системы.

В Oberon-2 для повышения удобства и ясности кода была введена специальная синтаксическая конструкция — связанные процедуры.

Пример 2. ООП в Oberon-2 [2].

```
TYPE
    Figure = RECORD
        color: INTEGER
    END;
    Rectangle = RECORD (Figure)
        x0, y0, x1, y1: REAL
    END;

PROCEDURE (VAR f: Figure) Square (): REAL
BEGIN
    RETURN 0
END Square;

PROCEDURE (VAR r: Rectangle) Square (): REAL;
BEGIN
    RETURN ABS(r.x1 - r.x0) * ABS(r.y1 - r.y0)
END Square;
```

В таком виде связанные процедуры полностью идентичны методам объектных языков программирования, с их достоинствами и недостатками. Важно, что внутри связанных процедур работа с полями объекта выполняется через явно объявленный параметр.

В Component Pascal введены дополнительные модификаторы ООП, позволяющие явно выразить в коде архитектурные нюансы и повысить тем самым надежность компонентных систем. `EXTENSIBLE` — этот модификатор требуется явно указывать, чтобы разрешить расширение данного типа или процедуры. `ABSTRACT` — аналог `EXTENSIBLE`, запрещающий создание экземпляров типа, либо объявляющий сигнатуру связанной процедуры без тела, `EMPTY` — расширяемая связанная процедура с пустым телом. `LIMITED` — указывает на то, что создавать экземпляр типа может только модуль, в котором он объявлен, позволяет гарантировать, что для всех экземпляров типа выполнена требуемая

[2] См. Н. Вирт, Х. Мёссенбёк. Язык программирования Оберон-2. Посл. ред. июль 1996. Перевод на русский — С. Свердлов

инициализация. (Гораздо чаще, нежели LIMITED, используется сочетание абстрактного экспортированного типа и скрытого конкретного типа-реализации). NEW — маркирует процедуру, первый раз введенную в иерархии расширения, позволяет избежать опасной ошибки, когда в одном из базовых типов вводится связанная процедура, совпадающая с уже существовавшей процедурой в расширенном типе.

Пример 3. ООП в Component Pascal[3].

```
TYPE
    Figure = ABSTRACT RECORD
        color: INTEGER
    END;

    Rectangle = EXTENSIBLE RECORD (Figure)
        x0, y0, x1, y1: REAL
    END;

PROCEDURE (VAR f: Figure) Square (): REAL, NEW, ABSTRACT;

PROCEDURE (VAR r: Rectangle) Square (): REAL;
BEGIN
    RETURN ABS(r.x1 - r.x0) * ABS(r.y1 - r.y0)
END Square;
```

Как уже было сказано ранее, роль ООП в прикладном программировании заключается в непосредственном выражении в коде отношений сущностей из предметной области.

В системном программировании ООП несет архитектурную нагрузку. В суррогатных языках на классы возложена функциональность модулей, в частности, инкапсуляция. Однако в Оберонах, в которых присутствует полноценная отдельная синтаксическая конструкция модуля, на ООП также остается важная архитектурная нагрузка — обеспечение динамических связей между модулями. Триада «абстрактный тип — конкретная реализация — точка доступа (т.е. переменная — указатель абстрактного типа)» обеспечивает создание гибких разъемов между модулями, подключение которых в компонентных системах может выполняться динамически.

Пример 4. Абстрактный разъем между ядром и загрузчиком модулей в BlackBox

```
MODULE Kernel;
    ...
    TYPE
        LoaderHook = POINTER TO ABSTRACT RECORD (Kernel.Hook)
            res: INTEGER;
            importing, imported, object: ARRAY 256 OF CHAR;
            (h: LoaderHook) ThisMod (IN name: ARRAY OF SHORTCHAR):
                                                Kernel.Module, NEW, ABSTRACT
        END;
    PROCEDURE SetLoaderHook (h: LoaderHook);
```

[3]См. The Oberon Microsystems. Сообщение о языке Компонентный Паскаль. 1994-2001. Перевод на русский — Ф. Ткачев

```
    ...
END Kernel;
```

Модуль StdLoader инкапсулирует конкретную реализацию загрузчика и инсталлирует ее в ядро через разъем с интерфейсом `LoaderHook`. Ядро ничего не знает о конкретной реализации загрузчика. Таким образом, сводится к минимуму число статических зависимостей между модулями программной системы. Такой подход используется и в других языках — например, идиома Pimpl (Pointer to Implementation) в C++. Однако ее мощь проявляется именно в языках компонентного программирования, которые поддерживают динамическую загрузку модулей. Идея в том, что каждый из таких разъемов между модулями может быть переключен прямо во время выполнения приложения. Например, мы можем выполнить свою реализацию загрузчика модулей из сетевого репозитория, динамически загрузить модуль нового загрузчика и инсталлировать разъем от новой реализации в ядро, после чего станет возможной сетевая загрузка модулей, без перезапуска приложения.

В BlackBox часто используется один из подвидов таких объектно-ориентированных разъемов — инсталлируемые каталоги объектов, о чем будет сказано ниже.

В компонентных системах возникает важная проблема — проблема межмодульного наследования реализации. Подчеркнем, что эта проблема не проявляется при наследовании внутри одного модуля или внутри подсистемы, которая разрабатывается закрыто единой командой; она касается проектирования и реализации повторно используемых и широко распространяемых модулей, библиотек и особенно — целых компонентных каркасов (component framewrorks). Глубокие иерархии наследования, в которых присутствует наследование реализации, создают так называемую проблему хрупкого базового класса (fragile base class problem). Ее суть в том, что наследование реализации часто порождает сложные статические зависимости между поведением базового типа и типа-расширения. Таким образом, после публикации базового типа разработчик лишается возможности менять его реализацию из боязни нарушить работу типов-расширений, созданных сторонними разработчиками.

Еще одна проблема в компонентных системах — добавление в экспортированный тип новых полей либо методов (даже скрытых) потребует перекомпиляции всех модулей-клиентов. Это вполне нормально в рамках одной программной системы, но категорически неприемлемо для компонентного каркаса со множеством сторонних пользователей. Поэтому в компонентных системах экспортируются преимущественно абстрактные типы, конкретные же реализации скрыты, создаются фабричными процедурами и недоступны для расширения. В таких компонентных моделях, как COM и COBRA, наследование реализации вообще запрещено, опубликованными сущностями являются абстрактные интерфейсы, кроме того, изменение интерфейса после публикации строго запрещено. В Component Pascal разрешается и модификация опубликованных интерфейсов, и наследование реализации, однако при разработке компонентных каркасов для широкого использования придерживаются строго тех же правил — запрет и того, и другого.

Таким образом, для прикладного ООП характерны глубокие иерархии наследования, непосредственно отражающие взаимосвязи из предметной области; для технического ООП в компонентных системах характерны неглубокое наследование, состоящее из двух уровней — экспортированные абстрактные типы (которые могут наследоваться один от другого) и скрытые типы-реализации, которые наследуются исключительно от абстрактных типов-интерфейсов и создаются фабриками. Такое ограничение не

уменьшает, как может показаться на первый взгляд, а увеличивает гибкость систем — особенно при совместном использовании с родовыми шинами сообщений, о которых рассказано в следующем пункте. Необходимость такого подхода не сразу осознается при разработке отдельных приложений, этот подход родился на основе опыта создания операционных систем, в которых «отдельным приложением» является обычный модуль языка программирования, и термин «расширяемость» имеет более жесткое значение, чем принято иметь в виду: **«расширяемость» в Оберонах — это значит «динамическая расширяемость без перекомпиляции и перезапуска системы»**.

Еще одним приемом, использованным в среде BlackBox, является избежание неабстрактных расширяемых методов и супервызовов (явного вызова из расширенного в типе-потомке метода соответствующего метода базового типа). Этот прием применяется для повышения безопасности в тех случаях, когда все-таки используется наследование реализации. В объявлении связанных процедур рекомендовано использовать либо объявление `ABSTRACT` или `EMPTY`, либо конкретное нерасширяемое определение; `EXTENSIBLE` используется осмотрительно. Вместо `EXTENSIBLE` принято использовать версионированные процедуры-расширения, передающие вызов далее по цепочке. Например, тип `Containers.Container` является расширением типа `Views.View`, реализующим процедуру `HandleCtrlMsg`. Дальнейшее расширение этой процедуры невозможно, она не помечена как `EXTENSIBLE`. Вместо этого вводится пустая расширяемая процедура `Container.HandleCtrlMsg2`, которую `HandleCtrlMsg` вызывает в нужном месте своего кода. Дальнейшие типы-расширения `Container` определяют требуемую реализацию для `HandleCtrlMsg2`, таким образом косвенно расширяя базовую `Container.HandleCtrlMsg`. Если требуется ввести еще один уровень расширения, вводится еще одна версия «заглушек» — `HandleCtrlMsg3`. Достоинством этого подхода является большая безопасность в компонентных каркасах. Разработчик типа-расширения не может забыть или сознательно не вызвать базовую процедуру, или вызвать ее не из нужного места. Решение передачи управления расширенным процедурам принимается в процедурах базового типа, а не наоборот. В стандарте языка Component Pascal супервызовы помечены как устаревшее средство.

Сборка мусора является важным фактором для эффективного использования ООП. В компонентной системе со сборкой мусора каждый объект обладает самостоятельным жизненным циклом, не требуется специально реализовывать централизованных механизмов управления группами динамических объектов. В многопоточных приложениях этот принцип может быть расширен — не требуется централизованное управление потоками. Созданный активный объект может быть свободно выпущен в компонентную среду и будет выполняться до тех пор, пока на него есть ссылки. Как только ссылок не окажется, будет выставлен сигнал останова, и объект сможет завершиться. Такой механизм введен в Active BlackBox.

Важно также, чтобы переменные объектного типа могли быть статическими, а не только динамическими. В противном случае, если необходимо частое использование мелких короткоживущих объектов, то либо будет создаваться большая нагрузка на диспетчер памяти и сборщик мусора, либо разработчику придется отказываться от ООП. Такой недостаток, например, имеется в языке Java. Этот момент является важным для реализации родовой шины сообщений, описанной ниже.

2 Родовые шины сообщений

Родовые шины сообщений (generic message buses/interfaces) представляют собой механизм, позволяющий организовать в компонентной среде гибкое и расширяемое взаимодействие разнородных объектов, ничего не знающих друг о друге. Этот прием был использован Н. Виртом и Ю. Гуткнехтом в операционной системе Oberon и затем эффективно применялся в том или ином виде во всех Oberon-системах. Особенно ярко преимущества этого подхода проявляются в архитектуре графических интерфейсов.

Практически все графические интерфейсы базируются на понятии графического объекта — отображаемого на экране элемента. Графический объект в определенные моменты отрисовывает себя в требуемом месте экрана, а обратная связь с окружением осуществляется через обработку приходящих извне (от операционной системы, графической библиотеки и т.п.) сообщений. Концептуально графические интерфейсы являются объектно-ориентированными, однако на уровне API операционных систем работа обычно организуется процедурно. Операционная система «знает» сигнатуру и адрес обработчика сообщений, отвечающего за некоторый графический объект (например, окно), обработчик сообщений «знает» те сообщения, которое ему требуются, обрабатывает их должным образом и может просто игнорировать все остальные. Обычно сообщение — это жестко определенная структура, имеющая селектор типа сообщения, и дополнительные поля для параметров сообщения. Ключевой конструкцией обработчика сообщения является оператор выбора `CASE селектор_типа_сообщения OF...` Содержимое полей параметров может быть произвольным, и для извлечения данных обычно требуется обход типизации языка. Вся ответственность за корректное извлечение параметров и принудительное приведение типа в зависимости от значения селектора лежит на программисте, пишущем обработчик. Программирование в подобном стиле громоздко и неудобно, однако у него есть важное достоинство — такой интерфейс, основанный на сообщениях, легко расширяется. В новых версиях операционной системы могут вводиться новые сообщения, при этом не потребуется переписывать или перекомпилировать старые графические приложения. Напротив, некоторое приложение может ввести свои типы сообщений и использовать механизмы операционной системы для их передачи. При этом «осведомленные» графические объекты будут обрабатывать новые сообщения, а «несведущие» — просто игнорировать.

С широким распространением ООП стала очевидной разумность объектно-ориентированной работы с графическими интерфейсами. Идеи, использованные в графических библиотеках различных сред (например, Delphi VCL), примерно идентичны. Вводится базовый класс графического объекта, у которого есть перегружаемые методы, отвечающие за отрисовку объекта и, возможно, какие-то другие операции, и прототипы обработчиков различных событий — либо в виде перегружаемых методов, либо в виде полей — процедурных переменных, которые для конкретного экземпляра графического объекта могут быть связаны с конкретными обработчиками (т.е. используется ООП в стиле Oberon, что обеспечивает большую гибкость и динамическое связывание объекта с обработчиком). Этот в целом достаточно удобный подход имеет принципиальные ограничения. Здесь невозможно такое же гибкое расширения, как в механизме передачи сообщений. Для введения в систему нового вида событий требуется модифицировать один из общеиспользуемых, базовых типов графических объектов, вводя в него новые методы либо поля-слоты для подключения обработчиков. Это ведет к необходимости перекомпиляции всех разработанных ранее модулей. Поэтому такой подход применим

только в отдельно взятых средах разработки, на выходе которых получается статически собранное приложение, но не на уровне операционных систем или платформ выполнения (если только такая платформа не является интерпретирующей или использующей промежуточное представление кода). И наоборот — вводя в конкретном приложении новый тип событий, мы не можем использовать какие-либо общие механизмы системы для их передачи (например, широковещательной), иными словами, графическая среда и объекты других разработчиков являются непрозрачными для новых событий, вводимых нашим приложением. Этот изъян не так заметен при организации традиционных графических интерфейсов, структура которых очень проста — окно, на которое наложены графические объекты, иногда — промежуточные «подложки» в виде каких-либо панелей.

В то же время в средах Oberon принята идея составных документов — иерархий графических объектов произвольной вложенности. Объект-контейнер может ничего не знать о тех объектах, которые могут быть в него вложены, но он обязан обеспечить совершенно прозрачное взаимодействие вложенных объектов с верхним окружением. Те объекты в иерархии, которые «знают» друг о друге, могут целенаправленно взаимодействовать между собой, те объекты, которые «не знают» друг друга, должны прозрачно передавать необходимые сообщения от отправителей к получателям. Также сама графическая инфраструктура среды должна предоставить общий механизм взаимодействия для множества компонент сторонних производителей, о которых ничего неизвестно на этапе разработки и компиляции библиотек среды.

Н. Вирт, начиная проект OS Oberon, был прекрасно знаком с разработками лаборатории Xerox, где впервые был опробован оконный графический интерфейс. Передовые графические среды разрабатывались в Xerox на чистом объектно-ориентированном языке Smalltalk. Поэтому идея разработки графического интерфейса на ООП-концепциях была уже тогда очевидной. Oberon напоминает Smalltalk в плане наличия единой моноязыковой среды выполнения, которая динамически расширяется не изолированными приложениями, а непосредственно языковыми модулями или классами; границы между средой и пользовательскими приложениями полностью стерты. Однако Smalltalk — динамический интерпретируемый язык (позднее для него были разработаны сложные механизмы частичной предкомпиляции). Oberon же планировался как эффективный компилируемый язык, и перед разработчиками OS Oberon (Н. Вирт и Ю. Гуткнехт) стояла нетривиальная задача обеспечения гибкости, не уступающей интерпретирующим средам, подобным Smalltalk. Такую гибкость дало сочетание динамической загрузки модулей, ООП, сборки мусора и полной метаинформации о модулях и типах на этапе выполнения. Однако проблема расширения базовых типов в компилирующих средах неизбежна. Однажды опубликованный объектный интерфейс не может быть изменен без того, чтобы не повлечь за собой перекомпиляцию всех клиентов. Поэтому «наивное» ООП с введением для каждого события своего метода либо поля-слота и глубокими иерархиями наследования оказалось непригодным по причинам, описанным выше.

Естественным решением оказался возврат к механизму сообщений, однако на новом, объектно-ориентированном уровне — введение родовой шины сообщений. Эта идея в той или иной форме используется во всех Oberon-системах.

Все графические объекты, как обычно, являются расширением некоторого базового типа (в BlackBox — `Views.View`, называемый отображением). Каждый графический объект имеет один раз и навсегда зафиксированный обработчик вида:

```
PROCEDURE (v: View) HandleMsg (VAR msg: Message).
```

Иногда вводятся два или три обработчика — для различных категорий сообщений (в BlackBox — `HanldeCtrlMsg` — для всех управляющих сообщений, HandlePropMsg - для опроса графического объекта о его свойствах и предпочтениях, `HandleViewMsg` — для специфичных для объекта сообщений, касающихся его логического содержания).

Базовый тип Message является расширяемой записью, а все конкретные сообщения — его расширениями. Сообщения, являясь исключительно объектами данных, без поведения, могут вводиться на всех уровнях программной системы и образовывать сколь угодно сложные иерархии расширения — это никак не влияется на расширяемость системы и не вызывает описанной в предыдущем пункте проблемы хрупкого базового класса.

Каждый объект, отображаемый на экране, подключен к системной шине сообщений через процедурный разъем `HandleMsg`. Некоторое сообщение передается одному из экранных объектов (например, являющемуся фокусом). Объект, в зависимости от типа сообщения, может необходимым образом его обработать. Если тип сообщения объекту неизвестен, он его просто игнорирует. Если среди вложенных объектов контейнера есть подфокус, то сообщение может быть передано для обработки ему.

Базовых типов обычно несколько, но очень немного; среди них может использоваться наследование реализации. (В BlackBox определено несколько базовых типов для графических объектов, обладающих заготовками поведения: `Views.View`, `Controls.Control`, `Containers.Container`).

Система может неограниченно расширяться путем определения новых реализаций базовых типов и новых типов сообщений. При этом не требуется модификация либо перекомпиляция существующего кода, система расширяется «на лету».

Повышенная гибкость такого подхода объяснима формально. Как было сказано выше, ООП дает виртуализацию кода, т.е. принятие решений об обработке вызова метода откладывается до момента выполнения. Связывание вызова метода с конкретной реализацией метода производится в зависимости от динамического типа объекта. Родовая шина сообщений дает двойную виртуализацию вызова. Результат вызова `object.HandleMsg(message)` зависит от сочетания двух типов (`TYP(object)`, `TYP(message)`).

Как реализована обработка сообщений внутри `HandleMsg`? Примерно так же, как и в процедурном варианте, только вместо селектора по значению `CASE` используется оператор Оберона — селектор по типу:

```
WITH msg: MessageType1 DO
        (* здесь msg уже приведен к MessageType1 *)
    | msg: MessageType2 DO
        (* здесь msg уже приведен к MessageType2 *)
ELSE
    ...
END;
```

Как и в `CASE`, если ветка `ELSE` отсутствует, то в случае типа, не перечисленного явно, возникнет ошибка времени выполнения. Таким образом, становится невозможной «маскировка», «проглатывание» ошибок в логике, связанных с непредусмотренным вариантом.

Для эффективной реализации родовой шины сообщений требуется языковая поддержка, и Оберон ее предоставляет. Оператор `WITH` удобен тем, что выполняет одновременно как множественное сравнение, так и приведение типа переменной. Оператор `WITH` выполняется очень быстро. Каждая структура данных в Обероне имеет тег — указатель на описатель типа. Описатель содержит указатели на все базовые типы данного типа. Таким образом, проверка соответствия типов выполняется очень быстро — всего за одно сравнение указателей базовых типов соответствующего уровня. Для селектора `WITH` компилятор может построить эффективную таблицу переходов, аналогично `CASE`.

Также важно, что в Обероне объекты (расширяемые записи) могут создаваться как в динамической памяти, так и на стеке, и удобно передаваться по ссылке (в Component Pascal введены специальные вариации ссылочных параметров, для большей наглядности и надежности — `VAR`, `IN`, `OUT`). Таким образом, для передачи сообщений не требуется выделения динамической памяти, и шины сообщений работают очень эффективно, не поглощая память и не нагружая сборщик мусора.

Ни в одном из популярных индустриальных языков нет аналога оператора `WITH`, кроме того, есть и другие нюансы, которые препятствуют эффективному использованию идеи родовых шин сообщений.

В C++ организовать выбор по типу можно с помощью цепочки

```
if ( dynamic_cast<Type1>(msg) ) ...
else if ( dynamic_cast<Type2>(msg) ),
```

однако как сравнение `dynamic_cast`, так и цепочные проверки гораздо более медленны, чем `WITH`. Более того, `dynamic_cast` применим только к типам, имеющим виртуальную таблицу, таким образом, для каждого типа сообщений искусственно пришлось бы вводить пустой виртуальный метод, например, деструктор. Внутри каждой ветки пришлось бы выполнять явное преобразование к проверенному типу. Кроме того, до сих пор не все компиляторы поддерживают Run-Time Type Information в соответствии со стандартом, требуется включение специальных опций компиляции и т.п. То же самое касается Java и Delphi. Кроме того, в Java и Delphi объекты могут создаваться лишь в динамической памяти. Записи record в Delphi не поддерживают наследование и информацию о типе. В Java статически и стеково размещаемые структуры отсутствуют в принципе. Таким образом, все объекты сообщений могут создаваться и уничтожаться только динамически.

Архитектура GUI — хороший масштабный пример применения родовой шины сообщений. Однако идея родовых обработчиков сообщений вида `HandleMsg(VAR msg: Message)` применима и в менее глобальных случаях. При создании повторно используемых каркасов полезно в базовые абстрактные типы ввести родовой обработчик сообщений. Тогда в дальнейшем при необходимости стандартизировать некоторые дополнительные возможности не потребуется менять интерфейс базового типа (что недопустимо) или вводить расширенную его версию (что неудобно, т.к. пользователям придется постоянно проверять, какой именно из абстрактных интерфейсов поддерживается объектом), достаточно будет ввести сообщения для работы с новой функциональностью. Конкретные типы реализации могут поддерживать тот или иной набор сообщений, а неизвестные — просто игнорировать.

3 Инсталлируемые каталоги объектов

Инсталлируемые каталоги объектов (directories) являются одним из видов объектно-ориентированных межмодульных разъемов, про которые говорилось в пункте 1. Они позволяют решить проблему полного разделения интерфейсов и реализации каких-либо модулей каркаса и обеспечить замену реализации не только без перекомпиляции, но и прямо во время выполнения. Этот подход уже демонстрировался выше на примере взаимодействия ядра и загрузчика. Инсталлируемые каталоги представляют собой подобную, но более шаблонную и ориентированную на пользователей идею, которая применяется в компонентных каркасах.

Рассмотрим конкретный пример — модуль Files из BlackBox Framework. Этот модуль не содержит никакой платформенно-зависимой реализации. В нем объявлены абстрактные интерфейсы `File`, `Reader` и `Writer` для работы с файлами, `Locator` — для работы с каталогами. Платформенно-зависимая реализация этих типов вынесена в отдельный модуль HostFiles подсистемы Host и скрыта от разработчиков, которые работают с файлами исключительно через модуль Files и его типы. Однако каким-то образом должны создаваться экземпляры конкретных типов, реализующих эти интерфейсы. Для этих целей в интерфейсном модуле Files объявлен тип `Directory` — тип-фабрика, в данной модели называемый каталогом, создающий экземпляры объектов типа `File`, `Locator` — либо для новых, либо для уже существующих на диске файлов. Тип Directory также абстрактный. Также модуль Files экспортирует переменную только для чтения `dir`, имеющую статический тип `Directory`, и процедуру `SetDirectory`, позволяющую инсталлировать в модуль конкретную реализацию каталога. В сокращенном виде интерфейс модуля Files выглядит следующим образом:

```
DEFINITION Files;

    TYPE
        Name = ARRAY 256 OF CHAR;

        Type = ARRAY 16 OF CHAR;

        File = POINTER TO ABSTRACT RECORD
            ...
            (f: File) Close, NEW, ABSTRACT;
            (f: File) NewReader (old: Reader): Reader, NEW, ABSTRACT;
            (f: File) NewWriter (old: Writer): Writer, NEW, ABSTRACT;
            ...
        END;

        Reader = POINTER TO ABSTRACT RECORD
            eof: BOOLEAN;
            (r: Reader) Pos (): INTEGER, NEW, ABSTRACT;
            (r: Reader) ReadByte (OUT x: BYTE), NEW, ABSTRACT;
            (r: Reader) SetPos (pos: INTEGER), NEW, ABSTRACT
            ...
        END;

        Writer = POINTER TO ABSTRACT RECORD
```

```
            (w: Writer) Pos (): INTEGER, NEW, ABSTRACT;
            (w: Writer) SetPos (pos: INTEGER), NEW, ABSTRACT;
            (w: Writer) WriteByte (x: BYTE), NEW, ABSTRACT
            ...
        END;

        Locator = POINTER TO ABSTRACT RECORD
            res: INTEGER;
            (l: Locator) This (IN path: ARRAY OF CHAR): Locator, NEW, ABSTRACT
        END

        Directory = POINTER TO ABSTRACT RECORD
            (d: Directory) New (loc: Locator; ask: BOOLEAN): File, NEW, ABSTRACT;
            (d: Directory) Old (loc: Locator; name: Name; shared: BOOLEAN):
                                                              File, NEW, ABSTRACT;
            (d: Directory) This (IN path: ARRAY OF CHAR): Locator, NEW, ABSTRACT
            ...
        END;

    VAR
        dir-: Directory;

    PROCEDURE SetDir (d: Directory);

END Files.
```

При запуске среды загружается модуль реализации HostFiles, который инсталлирует в Files реализацию каталога, работающего с файлами на конкретной платформе. Программисты работают с файлами, создавая объекты `Locator` и `File` через объект-каталог `Files.dir`.

Все модули BlackBox Framework не содержат платформенно-зависимого кода. Таким образом, все модули пользователей статически зависят только от платформенно-независимых модулей. Модули реализации из подсистемы Host не импортируются ниоткуда и могут быть переписаны под целевую платформу. При этом все написанные ранее пользовательские модули будут работать на новой платформе без изменений. А если среда портируется лишь в новую операционную систему на той же аппаратной платформе, то пользовательские модули и модули каркаса не потребуют даже перекомпиляции. В частности, разрабатывающаяся версия BlackBox под Linux использует тот же формат двоичных модулей, что и версия под Windows, и не требует перекомпиляции, если только модули напрямую не работают с операционной системой.

Продолжая пример с файлами, заметим, что механизм инсталлируемых каталогов позволяет динамически задействовать поддержку любого требуемой реализации файловой системы. Например, в стандартную поставку BlackBox входят модули DevPacker и StdPackedFiles. Паковщик позволяет создать в конце exe-файла среды виртуальную файловую систему с произвольными файлами. В один exe-файл можно поместить все кодовые файлы модулей и файлы ресурсов (заметим, что в отличие от статической линковки DevLinker, упакованные модули будут грузиться по-прежнему динамически, только хранятся в виде ресурсов внутри исполняемого файла). Работа с упакованными файлами происходит совершенно прозрачно. через тот же модуль Files. Модуль

StdPackedFiles инсталлирует каталог, поддерживающий работу с упакованной файловой системой. В случае, если запрашиваемый файл отсутствует в ней, StdPackedFiles перенаправит вызов к сохраненному стандартному каталогу. Аналогичным образом, абсолютно прозрачно, можно реализовать работу с шифрованными файлами, с сетевыми хранилищами и т.п.

Инсталлируемые каталоги используются не только для отделения платформенно-зависимой или специфической реализации, в BlackBox их принято использовать практически всегда, если предполагается широкое использование создаваемого модуля, и его типы не являются элементарными. Подобная гибкость оправдывает себя при длительной эволюции и модификации систем.

Например, работа с текстовыми отображениями-контейнерами обеспечивается модулями TextViews, TextModels, TextControllers. Эти модули экспортируют только абстрактные типы-интерфейсы и инсталлируемые каталоги. Поскольку реализация текста не является платформенно-зависимой, она не выносится в отдельные модули, а реализуется прямо там же, но инкапсулированно. Если открыть исходный код TextViews, то мы увидим экспортированные типы View, Directory и внутренние — StdView, StdDirectory. Модулям-клиентам незачем привязываться к конкретной реализации текста, все, что им нужно знать — это абстрактный интерфейс. Если мы захотим сделать свою реализацию текста, мы сможем инсталлировать новый каталог, и вся среда начнет совершенно прозрачно работать с новой реализацией.

В компонентных системах необходимо сводить к минимуму статические зависимости модулей от нюансов реализации друг друга — это обеспечивает расширяемость и взаимозаменяемость компонент системы.

4 Выгрузка и перезагрузка модулей в компонентных системах

Идея компонентного программирования, зародившаяся в Оберон-системах, получила широкое распространение в таких технологиях, как Java и .NET, в которых введена динамическая загрузка «модулей» — классов в Java и двоичных сборок в .NET. Однако роль выгрузки и перезагрузки модулей в компонентных системах до сих пор недооценена. То, что такая возможность для Oberon-систем является неотъемлемой, объясняется тем, что изначально Обероны — это операционные системы, в которых модули — эквиваленты приложений. Невозможно представить себе операционную систему, в которой нельзя выгрузить и перезапустить приложение. Среда BlackBox, работая поверх другой операционной системой, сохранила все качества мини-ОС, являясь интегрированной средой разработки и выполнения, в которой нет границы между самой средой и разрабатываемыми приложениями. Полноценная динамическая модульность позволяет выполнять многие «фокусы», которые невозможны ни в одной другой компилирующей среде. В частности, это касается работы с составными документами, поддерживающими произвольное комбинирование различных графических объектов, удобной разработки графических интерфейсов, когда редактируемое окно привязывается к модулю приложения «на лету» и параллельно в одной среде открыто для синхронного редактирования и тестирования, и т.п. Разработка среды, подобной BlackBox, на .NET, была бы связана со многими сложностями в реализации динамической выгрузки и перезагрузки модулей.

Бесспорно, динамическая выгрузка и замена произвольных модулей компонентной системы связана со многими проблемами. Подсистема времени выполнения может проверить отсутствие в памяти модулей, импортирующих выгружаемый, однако отследить наличие динамических связей гораздо сложнее. В памяти могут существовать объекты типов выгружаемого модуля. В BlackBox после выгрузки модуля в памяти остается секция описаний типов, таким образом, работа с данными типов модуля возможна и после его выгрузки. В памяти могут существовать одновременно много версий одного и того же модуля, однако секции глобальных переменных и кода присутствуют только для одной из них, загруженной позднее всех. Однако при вызове метода объекта или обращению по процедурной переменной, указывающей на процедуру выгруженного модуля, в BlackBox возникает ошибка времени выполнения. Естественно, что для среды любые ошибки подобного рода безопасны и не могут нарушить ее работы, однако работа пользовательского кода может быть нарушена. В частности, могут перейти в инвалидное состояние графические объекты на экране, модули которых были выгружены.

Поскольку в BlackBox присутствует полная метаинформация о модулях и типах на этапе выполнения, то не представляет сложности реализовать перед выгрузкой модуля проверку на отсутствие в памяти объектов его типов, обходя глобальные переменные, кучу и стек аналогично тому, как это делает сборщик мусора. Вероятно, в следующих версиях среды такая возможность появится.

Однако такой контроль — только полдела, он может лишь предотвратить небезопасную выгрузку модуля, но не может помочь организовать динамическую замену используемого модуля. Развивая идею дальше, можно реализовать такую замену следующим образом: выполнение приложения приостанавливается, в памяти ищутся все экземпляры объектов типов модуля и указатели на его процедуры, модуль выгружается, загружается новая версия, по интерфейсам полностью совместимая со старой, все указатели на старые объекты «изымаются» и заменяются указателями на вновь созданные экземпляры соответствующих типов. Необходим механизм копирования состояния старых объектов в новые. Для этого может быть использована, например, сериализация в двоичный поток либо специальный копирующий интерфейс, по образцу модуля Stores. Реализовать полностью автоматическую перезагрузку любого модуля не представляется возможным, модуль должен быть разработан с поддержкой такой операции, в частности — сериализации состояния своих объектов.

Механизм динамической перезагрузки модулей компонентных систем представляется весьма перспективным направлением для исследований.

Примечание редактора. *Все упомянутые в ссылках материалы доступны на сайте* `http://blackbox.metasystems.ru/`