Active BlackBox - документация разработчика

5 Механизм *деталей* - автоагрегация вместо множественного наследования

5.1 Множественное наследование или агрегация?

Со времени появления ООП темой для споров является множественное наследование и его необходимость в объектно-ориентированных языках. Классический ОО-язык Smalltalk не допускал множественного наследования. В гибридном языке *C++* оно есть в полном варианте. Сегодня принято выделять две разновидности множественного наследования: наследование реализации и наследование интерфейсов. Наследование реализации сопряжено со множеством проблем, которые могут возникать как при его использовании разработчиками, так и тех, которые приходится решать разработчикам компилятора. Поэтому общепринятым стало мнение о вреде множественного наследования реализации, и в новых языках программирования его обычно нет (*Java, C#*), присутствует лишь наследование интерфейсов. В языках Паскаль-семейства множественное наследование отсутствует. Однако в *Active Oberon* есть множественное наследование интерфейсов (*DEFINITIONs*). Также множественное наследование интерфейсов присутствует в *Delphi* (*interfaces*
).
Наследование (которое в *Oberon* называется расширением типа) предназначено для выражения отношений "вид-род", "все *A* есть *B*", что подразумевает под собой "*A* может использоваться всюду, где и *B*". Прикладная роль наследования заключается в возможности выразить средствами языка классификацию сущностей предметной области. Техническая его роль заключается в обеспечении расширяемости программной системы без переписывания и перекомпиляции старого кода. Расширяемые типы данных позволяют

создавать гибкие разъемы между модулями программной системы, т.е. обеспечивать удобные и общие *интерфейсы* для системного программирования. В Оберон-системах и в *BlackBox Framework* расширяемые типы используются в первую очередь таким образом. Используются в основном мелкие иерархии расширения - обычно экспортируется абстрактный тип - интерфейс, под который создаются конкретные неэкспортируемые типы-реализации.

Идея множественного наследования может происходить из соображений трех планов. Во-первых, с прикладной точки зрения, любой объект может входить в несколько независимых классификаций, т.е. являться одновременно расширением нескольких типов. Во-вторых, один объект реализации может поддерживать несколько различных разъемов-интерфейсов. И в третьих, один объект может являться композицией, обладать внутренними возможностями сразу нескольких типов.

Соображения первого рода на практике проявляются очень редко и обычно легко разрешаются без множественного наследования путем более тщательного анализа предметной области.
Соображения второго и третьего рода можно разрешить путем использования *агрегации* вместо наследования. Т.е. иерархия наследования задействуется только под главную связь, остальные реализуются через введение объектов базовых типов членами расширяющего типа. В случаях второго рода - при поддержке нескольких интерфейсов - при реализации типа требуется дополнительная ручная работа по написанию конкретных разъемов под каждый интерфейс, а при его использовании - немного более длинная форма записи, с обращением к требуемому разъему. В этом случае наличие в языке наследования интерфейсов может повысить лаконичность и наглядность кода, однако его отсутствие также не является серьезным недостатком. В третьем же случае никаких преимуществ перед агрегацией множественное наследование реализации вообще не дает. Одним из недостатков агрегации как замены наследованию интерфейсов является невозможность средствами языка привести переменную базового типа к агрегированному в ее фактическом типе интерфейсу или просто узнать, поддерживается ли этот интерфейс фактическим типом переменной. Однако для языковых сред, поддерживающих метаинформацию времени выполнения, это становится возможным. В частности, ядро *BlackBox* предоставляет для каждого типа полную информацию о его полях и их типах. Однако для удобного извлечения агрегированных интерфейсов желательно иметь специализированную группу процедур. Таким образом мы приходим к идее введения в подсистему времени выполнения некоторого механизма, позволяющего работать с агрегированными в типе интерфейсами, к идее *управляемой* агрегации.

Реализация подобных механизмов на основе системы времени выполнения, без расширения языка, не является чем-то новым для Оберонов. Язык изначально задумывался как предельно компактный, эффективно расширяемый за счет библиотек, для чего впервые среди компилируемых языков была введена поддержка всей метаинформации о структурах программы на этапе выполнения.

5.2 Автоагрегация - компонентность на уровне типов данных

Рассмотрим конкретный пример - реализацию активных объектов модулем *Ao* (предполагается, что читатель уже ознакомился с соответствующими главами документации). Модуль экспортирует служебный тип *MONITOR*, объявив который полем нашего типа, мы делаем его монитором и можем создавать для него эксклюзивные процедуры и использовать операцию *Ao.AWAIT*. То есть, имеем схему: *модуль-сервис* предоставляет некоторый *механизм*, который может интегрироваться в типы данных модулей-клиентов, после чего модуль-сервис обеспечивает для этих типов некоторое *обслуживание*. Назовем служебный тип для некоторого механизма *деталью*. Пусть клиентский тип-агрегат *A* использует служебный тип - деталь *D* модуля-сервиса *S*. Тогда всю схему назовем *S-D-A* (сервис-деталь-агрегат).

Часто, при поверхностном проектировании, схема *S-D-A* реализуется через наследование пользовательских типов от типа-детали. Однако такой подход имеет большой изъян - в языках с одиночным наследованием иерархия расширения целиком захватывается второстепенной связью конкретного пользовательского типа, который может уже являться по своему смыслу расширением какого-либо базового или даже абстрактного типа, с конкретным служебным механизмом, который является всего лишь деталью реализации. Наличие множественного наследования интерфейсов в языке в данном случае не помогает, т.к. тип-деталь является конкретным типом с реализацией. Поэтому единственно приемлемым вариантом в данном случае является агрегация. Агрегация позволяет одному типу-клиенту интегрировать в своей реализации

несколько механизмов нескольких сервисных модулей, т.е. использовать много деталей.

Деталь выполняет функцию информационного тега для сервисов *S*. *D* интегрируется через агрегацию, т.е. *A* имеет поле типа *D*. Здесь есть две ключевые проблемы. **Во-первых**, для корректной работы после создания экземпляра *A* требуется инициализация состояния *D*, кроме того, в большинстве случаев *D* должна иметь обратную связь с *A*, то есть, деталь должна знать агрегат, в который она интегрирована. Единственный способ это сделать - явно выполнять инициализацию детали, вызывая после создания экземпляра *A* инициализационную процедуру *S.Init(A, D)*. Такой вызов в *Component Pascal* должна явно выполнять фабричная процедура *A*. Это требование создает серьезные неудобства при использовании нескольких деталей, особенно на нескольких уровнях наследования. Отметим, что наличие в языке классических конструкторов (для модульного языка, в принципе, избыточных) не изменило бы ситуации, так как конструктор детали *D* должен был бы каким-либо образом получить ссылку на свой агрегат, то есть, потребовался бы его явный вызов из конструктора *A*
.
Во-вторых, сервисам модуля *S* требуется получать доступ к сооветствующим деталям обслуживаемых экземпляров *A*. Единственный способ это сделать - при обращении к *S* передавать непосредственно ссылку на поле типа *D*. Это ведет к загромождению кода и ошибкам, особенно при наличии многих деталей на многих уровнях расширения типа. В случае использования наследования информацию о базовых типах конкретного экземпляра всегда можно получить с помощью средств языка и/или *RTTI*
. В случае агрегации языковые средства для этих целей отсутствуют. В Оберон-средах получить информацию о структуре типа возможно с помощью метамеханизмов, однако для быстрого доступа к деталям этот способ слишком громоздкий и неэффективный.
Резюмируем вышесказанное. Для эффективной реализации модели *S-D-A*
необходимо наличие следующих средств языка либо системы времени выполнения:
1) В момент создания экземпляра *A* должно выполняться автоматическое создание всех деталей типа, с вызовом методов-инициализаторов деталей *D.Init(A)*
, которым передаётся ссылка на объект-агрегат. То есть, должна выполняться автоматическая интеграция деталей и объекта-агрегата.

2) Для каждого типа должен поддерживаться дескриптор агрегируемых деталей. Высокоуровневые процедуры среды должны позволять удобно получить информацию о поддерживаемых объектом деталях и ссылку на деталь конкретного типа.
Такой механизм можно назвать *автоматический агрегацией*. Автоагрегация является очередным шагом в поддержке компонентного программирования, позволяя компонентам-сервисам автоматически интегрироваться с пользовательскими типами данных. То есть, облегчается задача сборки пользовательских типов данных из предоставляемых сторонними компонентами деталей. Автоагрегация распространяет компонентный подход от динамической интеграции модулей к динамической интеграции типов данных в схеме *S-D-A*
. Побочный эффект от п. 2 автоагрегации - возможность эмуляции множественного наследования интерфейсов в языке, в котором его нет. Побочный эффект от п. 1 автоагрегации - возможность эмуляции конструкторов в языке, в котором их нет (однако весьма сомнительна практическая польза от этого за исключением некоторых системно-библиотечных задач).
Схема *S-D-A* харатеркна для механизмов системного назначения - например, библиотек параллельного программирования, перманентных и распределенных объектов и т.п. Реализовывать модули-сервисы рядовому программисту приходится редко, но использовать в приложениях - часто. Поэтому от продуманности реализации модулей-сервисов и способа интеграции типов-деталей в пользовательские типы зависит удобство работы прикладного программиста. Именно поэтому, исходя из практических задач, нами был рзработан runtime-механизм *автоагрегации деталей*. Он позволяет реализовывать системные библиотеки, подобные модулю *Ao*
, которые просты для использования настолько, как если бы данные средства были введены непосредственно в язык. В сочетании с метамеханизмами среды это позволяет выполнять многие системные расширения без изменения языка и компилятора.

5.3 Механизм *деталей* в *Active BlackBox*

Автоматическая агрегация для динамических объектов в *Active BlackBox* поддерживается на уровне ядра, без каких-либо изменений в компиляторе, с сохранением полной обратной совместимости с обычной версией *BlackBox*. Для создания автоагрегируемых деталей используется базовый указательный тип *Kernel.Detail*. При создании экземпляра некоторого типа-агрегата *NEW*(*указатель на агрегат*) будут автоматически созданы все его детали и выполнена их инициализация - вызов процедуры *Detail.Init(указатель на агрегат*).

Для каждого типа данных, загруженного в среде, который имеет поля-детали, ядро создает дескриптор деталей, позволяющий получать быстрый доступ к деталям динамических экземпляров этого типа. Ядро предоставляет низкоуровневые процедуры для получения такой информации, не взирая на инкапсуляцию деталей, т.е. наличия/отсутствия для поля метки экспорта (см. документацию ядра). Использование этих процедур непосредственно из кода ваших модулей не рекомендуется. Отметим, что поддержка деталей введена в ядро максимально эффективно и не вносит накладных расходов для обычных типов, не имеющих дескриптора деталей.

Для использования деталей в пользовательских модулях введен модуль *Details*, являющийся высокоуровневой оболочкой для средств ядра. В нем определен псевдоним для типа *Kernel.Detail* и процедура *DetailsOf*

, позволяющая получить указатели на интересующие детали объекта. При этом учитывается метка экспорта поля детали - получить доступ к скрытой детали невозможно. Отметим, что сам тип-агрегат может не экспортироваться, метку экспорта должно иметь его поле-деталь.

Приведем примеры использования модуля *Details(* Листинг 5.1 Эмуляция конструктора *)*

```
MODULE ObxDetailsCon;

  IMPORT Details, Log := StdLog;

  TYPE
    Constructor = POINTER TO RECORD (Details.Detail) END;
    Object = POINTER TO RECORD
      con: Constructor;
      a, b, c: INTEGER
    END;

  VAR
    object: Object;

  PROCEDURE (c: Constructor) Init (obj: ANYPTR);
  BEGIN
    Details.AssertSingle(obj, c);
    WITH obj: Object DO
      obj.a := 1; obj.b := 2; obj.c := 3
    END
  END Init;

  PROCEDURE Test* ;
  BEGIN
    NEW(object);
    Log.Int(object.a); Log.Tab;
    Log.Int(object.b); Log.Tab;
    Log.Int(object.c); Log.Ln
  END Test;

END ObxDetailsCon.
```

```
DevCompiler.CompileAndUnload
ObxDetailsCon.Test
```

Откомпилируйте модуль *ObxDetailsCon* и выполните команду *Test*. При создании экземпляра *Object* будет создана его деталь *Constructor*
, которая выполнит инициализацию полей своего агрегата, в чем можно убедиться по выводу в окошко протокола.
Строка *Details.AssertSingle* выполняет проверку единственности детали данного типа у объекта. Ядро инициализирует детали в порядке от базовых типов к расширениям. Метод *Init* детали вызывается сразу после ее создания, но до ее записи в поле объекта-агрегата. Таким образом, *AssertSingle*
проверяет отсутствие у объекта-агрегата деталей данного типа до момента создания текущей детали и позволяет обнаруживать ошибки повторного включения в расширенных типах тех деталей, которые уже были включены в одном из базовых типов, - для тех деталей, которые не позволяют своей многократной агрегации.
(Листинг 5.3 Использование шпиона *)*

```
MODULE ObxDetailsSpyTest;

  IMPORT Spy := ObxDetailsSpy, Ao, Services, Dialog, Strings;

  TYPE
    Generator* = POINTER TO RECORD
      f0, f1, f2: INTEGER;
      tag: Ao.MONITOR;
      spy: Spy.Spy;
      stop: Ao.Stop
    END;

  VAR
    anchor: Generator;

  PROCEDURE (g: Generator) GetCurrent* (OUT f0, f1, f2: INTEGER), NEW;
  BEGIN Ao.EXCLUSIVE;
    f0 := g.f0; f1 := g.f1; f2 := g.f2
  END GetCurrent;

  PROCEDURE Fibbonachi (g: Generator);
  BEGIN Ao.EXCLUSIVE;
    g.f2 := g.f2 + g.f1;
    g.f1 := g.f2 - g.f1;
    g.f0 := g.f2 - g.f1
  END Fibbonachi;

  PROCEDURE Gen (g: Generator);
  BEGIN Ao.ACTIVE(Ao.bound);
    g.f0 := 1; g.f1 := 1; g.f2 := 2;
    WHILE ~g.stop.ShouldStop() DO
      Fibbonachi(g);
      Ao.Sleep(1000)
```

```
    END
  END Gen;

  PROCEDURE Test* ;
  BEGIN
    NEW(anchor);
    Gen(anchor)
  END Test;

  PROCEDURE Collect* ;
  BEGIN
    anchor := NIL;
    Services.Collect
  END Collect;

  PROCEDURE CaptionGuard* (VAR par: Dialog.Par);
    VAR s: ARRAY 16 OF CHAR;
        f0, f1, f2: INTEGER;
  BEGIN
    IF anchor # NIL THEN
      anchor.GetCurrent(f0, f1, f2);
      Strings.IntToString(f0, s);
      par.label := s$ + ", ";
      Strings.IntToString(f1, s);
      par.label := par.label + s$ + ", ";
      Strings.IntToString(f2, s);
      par.label := par.label + s$
    ELSE
      par.label := "NIL"
    END
  END CaptionGuard;

END ObxDetailsSpyTest.

DevCompiler.CompileAndUnload
ObxDetailsSpyTest.Test
ObxDetailsSpyTest.Collect
```

И в завершение приведем пример сервисного модуля, предоставляющего деталь-шпион, которая с интервалом 5 секунда выводит в протокол информацию о своем агрегате и состояние его экспортированных полей.

Запустите команду *Test*

- будет создан экземпляр тестового объекта, три поля которого содержат очередные члены последовательности Фиббоначи, динамически генерируемые активной процедурой. Деталь-шпион будет выводить в протокол изменяющееся состояние объекта. Обратите внимание на реализацию экслюзивного доступа к объекту в процедуре *ObxDetailsSpy.Watch*

. Процедура ничего не знает об объекте, с которым рабоает, поэтому проверяет поддержку детали-монитора. Если объект является монитором, то работа с ним ведется в эксклюзивном режиме - это пример внешней эксклюзивной процедуры для активного объекта.

По команде *Collect* глобальный якорь на объект будет обнулен, активности объекта и детали остановлены, объект и деталь удалены сборщиком мусора.