Неудачные решения в Delphi

Гумеров Максим Маратович
Инновационно-аналитический отдел
ООО «Уфимский НТЦ»
Уфа, Российская Федерация
mgumerov@gmail.com

Since its acquisition by Embarcadero Technologies, Delphi is being renovated constantly by introducing some new features in an attempt to reduce the gap to more modern languages like C#. This tendency could quite probably strengthen Delphi's position as a development tool for startups.

The present paper relies on the author's experience with a variety of Delphi versions for Win32, from Borland Delphi 3.0 to Embarcadero Delphi XE, and it summarizes a research aimed at identifying some inadvisable development patterns, as well as suggesting more appropriate techniques.

The research studies both few bad design solutions in development and a number of malfunctions and inconsistencies in Delphi subsystems. For former ones, there will be a reasoning why they are better avoided, and how; for latter ones, some workarounds will be proposed.

Areas of interest include usage of OOD interfaces in Delphi as a means of introducing abstractions, event notification, source code version control and building projects with MSBuild. The paper shows some pitfalls with interfaces, like switching to interface-based conditional logics, or overusing downcasting instead of some type-safe way of querying interfaces, and how it makes applying some design patterns more complicated. It also discusses VCL routine ProcessMessages, and a very annoying bug rendering Delphi debugger nearly unusable.

Delphi; interface; OOD; event; wrapper; aggregation; resources; VCS; MSBuild

После приобретения прав на продукт Delphi компанией Embarcadero Technologies, примерно с 2009 года предпринимаются попытки актуализировать язык, внедрив в него возможности, недостаток которых стал ощущаться особенно сильно. Вероятно, эта совокупность нововведений укрепит позиции Delphi в качестве средства разработки, выбираемого для запуска новых проектов.

В основе данной статьи лежит опыт работы с версиями Delphi (для Win32), от Borland Delphi 3.0 и по Embarcadero Delphi XE, и проведенное автором исследование, имевшее целью обозначить некоторые трудности, которые могут встретиться при разработке новых проектов, и неудачные решения, которых по определенным причинам следует избегать, а также дать рекомендации по выбору более желательных альтернатив.

Delphi; интерфейс; ООП; событие; обертка; агрегат; ресурс; VCS; MSBuild

I. ВВЕДЕНИЕ

В последние 3-5 лет в сфере создания программного обеспечения очевидно явное переключение внимания разработчиков с бывшего ранее традиционным программного обеспечения персональных компьютеров на онлайн-сервисы, создание сайтов, разработку приложений для мобильных платформ (имеются в виду мобильные телефоны и планшеты). Причин тому немало – начиная от того банального факта, что это нынешний тренд, и того, что тренд этот всё еще сохраняет эффект новизны, умело подпитываемый социальными сетями и производителями сотовых телефонов и планшетов, а новизна привлекает в эту область молодых разработчиков, что и определяет сферу интересов новых запускаемых проектов, и заканчивая экономическими и техническими соображениями, такими, например, как легкость развертывания и массовая доступность подобного программного обеспечения, или эффект социальных сетей.

Что касается настольной разработки, бо́льшая часть проектов пишется на C++, Delphi (и других диалектах Паскаля), C# и Java. Сложно оценить долю проектов, пишущихся на каждом из этих языков, поскольку семейство проектов на Java включает не только лишь "настольные" приложения, но и огромное количество онлайн-проектов и приложений для мобильных устройств, C#, аналогично, применяется также и для онлайн-сервисов. Как бы там ни было, в тех случаях, когда создание кросс-платформного приложения не является целью, еще несколько лет назад были причины запускать проект на Delphi. Последовавший период, в который развитие языка Delphi было практически заброшено, существенно уменьшил актуальность этого языка при запуске новых проектов. Однако, после приобретения прав на продукт Delphi компанией Embarcadero Technologies в 2008 году, примерно с 2009 года предпринимаются активные попытки актуализировать язык, внедрив в него возможности, недостаток которых стал ощущаться особенно сильно. Это, прежде всего, включает введение обобщенных типов, поддержки строк в кодировках Unicode, полноценной рефлексии. В версии Delphi XE предпринята очередная и довольно масштабная попытка сделать разработку для Delphi кроссплатформной, не исключая и мобильные платформы.

Вполне вероятно, эта совокупность нововведений укрепит позиции Delphi в качестве средства разработки, выбираемого для запуска новых проектов. Автор далек от намерений анализировать, достоин ли этого данный язык и в каких проектах следует его выбирать. Цель данной статьи – дать срез своего опыта работы с версиями Delphi

(для Win32), от Borland Delphi 3.0 и по Embarcadero Delphi XE, в виде перечня некоторых трудностей, которые могут встретиться при разработке новых проектов, и неудачных решений, которых по определенным причинам следует избегать, и по возможности предложить соответствующие решения и пути обхода.

II. Проблемы интерфейсов как инструмента абстракции

В целях извечной тяги разработчиков компилятора Delphi к оптимизации там, где она лишь мешает, восходящее приведение (upcast) интерфейсов, т.е. переход от интерфейса-потомка к интерфейсу-предку, выполняется не при помощи вызова QueryInterface, а простым "усечением" (с точки зрения компилятора) интерфейса-потомка до предка, т.к. таблица методов интерфейса-потомка начинается с таблицы предка. Иными словами, если IB унаследован от IA и есть переменные a:IA и b:IB, то a := b помещает в "a" тот же указатель, что находился в "b", минуя вызов QueryInterface. А последний может быть перекрыт в классе, возвращая, например, адрес временного конструируемого объекта-переходника (proxy), или адрес интерфейса другого объекта (агрегированного), или демонстрируя еще какое-либо неожиданное, но в принципе корректное поведение. И даже если это не так, объект, на который указывает b, может реализовывать интерфейс IA дважды (простой пример: все интерфейсы наследуют IUnknown, значит, если класс реализует два разных интерфейса I1 и I2, ни один из которых не является потомком другого, то и таблица методов для I1, и таблица для I2 начинаются с таблицы методов IUnknown. Теперь если есть переменные unk: IUnknown, ip1: I1, ip2: I2, то присваивания unk:=ip1 и unk:=ip2 дадут в unk разные указатели. А должны, согласно требованиям COM, давать один и тот же! И большинство подпрограмм рассчитывают на это. Например, поиск элемента в IInterfaceList принимает IUnknown и сравнивает его с уже имеющимися в массиве IUnknown. С учетом усечения возможна ситуация, когда в массив поместили интерфейс ip1, усеченный до IUnknown, а потом пытаются искать тот же самый объект и тоже через IUnknown, но уже полученный из ip2, – и будет сделан вывод, что объекта в массиве нет.

Следует заметить, что в общем случае неверно ожидать, что QueryInterface при запросе для одного и того же интерфейса (у данного) возвращает всегда один и тот же указатель. COM выдвигает такое требование лишь к опросу интерфейса IUnknown. Документация по IUnknown::QueryInterface гласит: "For any one object, a specific query for the IUnknown interface on any of the object's interfaces must always return the same pointer value" (опрос IUnknown у любого интерфейса заданного объекта должен возвращать один и тот же указатель). Что означает, вдобавок, что сравнение интерфейсов всегда (даже если не брать в расчет возможность их усечения) следует проводить только после их приведения к IUnknown.

Также усечение интерфейсов отражается на допустимости присваиваний. Если IB наследует IA, то обычно поддержка неким объектом IB означает, что поддерживается и IA; но строго говоря, это не обязательно, и компилятор Delphi такого требования не выдвигает, и следовательно, можно определить объект obj, реализующий IB, но не IA. Приведение такого объекта к IA (obj as IA) приведет к ошибке. Но усечение интерфейсов позволяет обойти это правило: если объявлена var i: IA, то присваивание i := obj успешно выполнится, несмотря на отсутствие поддержки IA в obj.

Другая проблема, обусловленная бесконтрольной оптимизацией, состоит в объявлении функций, получающих интерфейс как const-параметр. Как и в случае с параметрами-строками, при такой передаче не увеличивается счетчик ссылок у интерфейса. Однако, авторы Delphi RTL не учли, что в отличие от строк, объекты, стоящие за интерфейсами, после их конструирования имеют в счетчике ссылок 0, а не 1. Это означает, что при вызове f(TMyObject.Create()), если procedure f(const instance: IMyObject) и TMyObject=class(IMyObject), переданный в f() интерфейс имеет нулевой счетчик ссылок. Delphi, как это ни парадоксально, несмотря на const, не препятствует внутри f изменять счетчик – например, присваивать instance другой переменной, или передавать в качестве параметра другому методу, который принимает этот параметр уже не как const. А в таком случае, если где-то внутри f счетчик увеличится на 1, а затем уменьшится на 1, и объект имеет обычную систему учета ссылок, вызывающую деструктор при достижении нулевого значения счетчика ссылок, то в результате объект будет разрушен. В контексте вызова f(TMyObject.Create()) было бы еще приемлемо, если бы экземпляр разрушался после возврата из f(), но в описанной ситуации он может разрушаться еще в процессе выполнения f(), например, в таком случае:

```
procedure f(const instance: IMyObject);
var copy: IMyObject;
begin
   copy := instance; //вызывает экземпляру _AddRef
   instance.DoSomething(); //работа с instance
   copy := nil; //вызывает экземпляру _Release

   //Теперь instance указывает на разрушенный
   //экземпляр; но мы в f() об этом не догадываемся
   //и пытаемся дальше с ним работать
   instance.DoSomething();
end;
```

Что интересно, инструкция вида f(TMyObject.Create()) приводит к проблеме только в том случае, если срабатывает описанное выше усечение интерфейсов. Если заблокировать его, переписав ее таким образом: f(TMyObject.Create() as IMyObject), то компилятор создаст для хранения результата приведения интерфейса временную переменную, при этом as увеличит счетчик ссылок, и в момент вызова f() он будет уже ненулевым.

Конечно, всегда доступно простое решение: выполнять приведение всякий раз, когда экземпляр передается в функцию. Это частично решает заодно и рассмотренную проблему усечения интерфейсов, но выглядит лексически и логически избыточно. Можно, с другой стороны, избегать объявления интерфейсов как const-параметров, и это еще удобнее, но функция, в которой три параметра const, а четвертый – нет, заставляет сомневаться о причинах этого. Еще одна интересная возможность

состоит в повсеместном следовании принципам внедрения внешних зависимостей (dependency injection): вместо создания в методе экземпляров класса передавать в этот метод фабрику, создающую такие элементы и возвращающую их интерфейсы. В этом случае можно создание почти любых экземпляров спрятать внутрь фабрик, а создание самих фабрик сосредоточить на верхнем уровне приложения, а то и вовсе автоматизировать. Тем самым, поскольку на instance к моменту вызова f(instance) уже имелась интерфейсная ссылка, уничтожение ему не грозит.

Ранее рассмотрены были трудности чисто технические. Опаснее связанная с интерфейсами идеологическая ловушка. Если для декларирования функциональности некой абстракции используется абстрактный класс, потомки могут определять конкретное поведение этих функций, их потомки – могут его менять, дополнять, но не могут притвориться реализацией другого набора функций, а не этого. Иными словами, никакой класс в таком случае не может иметь два разнородных (не связанных наследованием) "среза", или "проекции" (в виде других классов, в т.ч. абстрактных), хотя может иметь ровно одну (или несколько, которые являются потомками друг друга). Это означает, в частности, что в точке программы, где виден класс Class1, фактический класс объекта может быть любым потомком Class1, но не может быть потомком некоторого не связанного с ним Class2. Тот факт, что при использовании интерфейсов это возможно, порождает целое поле возможностей для злоупотреблений! Во-первых, очень соблазнительно становится в целях экономии времени возложить на один и тот же класс сразу несколько обязанностей, реализовав в нем несколько интерфейсов. Тогда, например, реализация абстракции "источник данных" может одновременно реализовывать абстракцию "источник событий", "структурированный объект", "объект с визуальными свойствами" и так далее. В лучшем случае это увеличивает связность между классами и нагрузку на класс, а высокие значения этих метрик ООП, как известно, ухудшают сопровождаемость проекта. Но подлинная глубина деградации архитектуры этим не исчерпывается. Если переходы между различными интерфейсами, которые реализует класс, явно отражены в интерфейсах, – либо же не отражены, но нет контракта на то, что эти интерфейсы реализованы одним и тем же объектом, – то такая архитектура неплохо читается и сопровождается. Если же допускаются отходы от этого принципа, программирование постепенно уходит в сторону условной логики путем опроса различных интерфейсов (case-программирования, если угодно). Отходы можно разделить на три категории:

1) "Маркерные" интерфейсы. В основном к этой категории относятся "пустые" интерфейсы, которые, на манер атрибутов в .Net, используются как управляющие воздействия на код, обрабатывающий объект, представленный интерфейсом, но не несущие какую-либо логику программы внутри себя. Например: if Supports(TheObject, IProgressable) then ProgressMonitor.Show(); {затем продолжается обработка объекта}

2) Утилитарные интерфейсы. Они предоставляют доступ к логике, ортогональной (в терминах Бертрана Майера [4], т.е. не связанной с) возможной бизнес-логике объектов, т.е. не связанной с ней. Это пересекается с идеей аспектно-ориентированного программирования, в которой аспектом называют специализацию класса путем привнесения в него логики, не связанной с его основным назначением (например, ведение записей в журнале событий при вызовах методов класса). Ярким представителем утилитарных интерфейсов является интерфейс IDisposable в .Net: практически всегда приходится только догадываться о том, следует ли данному экземпляру, сконструированному каким-то "черным ящиком" (фабрикой [1], например), вызывать Dispose(), когда он более не нужен: чтобы не пропустить нужный случай, нужно на всякий случай пытаться опрашивать поддержку IDisposable у всех подобных объектов. Что вообще-то порождает желание сделать этот интерфейс частью IUnknown.

3) Грани функциональности. Это случаи, когда две грани функций объекта, в принципе связанные друг с другом, на уровне контракта (соглашения) вынесены в два интерфейса, переход между которыми возможен только при помощи QueryInterface. Допустим, имеется интерфейс редактируемого документа, IDocument. А также – интерфейс "контейнер источников данных", IStorageContainer. Пусть в программе есть соглашение, по которому всякий документ является (или может являться) одновременно контейнером источников данных, но сам интерфейс IDocument при этом не имеет какого-то метода GetStorages(): IStorageContainer, а переход от doc: IDocument к IStorageContainer осуществляется выражением "doc as IStorageContainer". Вот это и есть пример условной логики на интерфейсах.

Первые два вида интерфейсов являются опциональными в том смысле, что работая с экземпляром, реализующим ISomething, нельзя знать заранее, поддерживает ли он также IDisposable, или нет (если он не является потомком IDisposable). Для маркерных эта особенность очевидна (иначе в них бы не было нужды), для утилитарных это тоже логично (коль скоро их логика ортогональна утилитарным функциям, странно было бы включать в их интерфейсы способы перехода к утилитарным интерфейсам). В третьем случае грани могут быть опциональны (A может иметь грань B) или обязательными (A непременно имеет грань B; C не имеет этой грани).

Чем же плохи опциональные интерфейсы? Тем, что конструкции вида if Supports(SomeInstance, IFlag) then .... и if Supports(Something, IChocolate, chocolate) then Eat(chocolate) являются вполне очевидными образцами условной логики, которую по ряду причин рекомендуется преобразовывать в полноценные ООП-решения на базе полиморфизма ([2], паттерн «Replace Conditional with Polymorphism»). Помимо прочего, при таком стиле кодирования логика обработки особенностей объектов выплескивается за пределы этих объектов, в обрабатывающий их код. И если для ортогональных интерфейсов это не столько плохо, сколько утомительно, то для маркерных и особенно для граней – это тревожный

признак. В перспективе сопровождение таких программ значительно затрудняется, а порог вхождения новичка в разработку - возрастает (как объяснить ему, что A может поддерживать B и C, а D нужно опрашивать у тех, кто поддерживает E, если из самого описания интерфейсов в коде этого не видно?)

Второй неприятной стороной применения опциональных интерфейсов является повышенная сложность создания объектов-оберток. Фактически, если накрыть оберткой или мостом обычный класс Delphi несложно без согласия последнего, то для мало-мальски успешного накрытия интерфейса необходимо либо точно знать, какие еще интерфейсы поддерживает накрываемый экземпляр, либо иметь с ним взаимную договоренность об агрегировании (см. далее раздел про обертки в Delphi) и надеяться, что неизвестные в данной точке программы (и потому не накрытые оберткой) методы не изменят функционирование экземпляра настолько, что какие-то из этих методов на самом деле следовало тоже обернуть.

Еще один источник неприятностей при работе с интерфейсами связан с процессами рефакторинга, это функции, принимающие нетипизированные параметры. Примером опасной функции такого рода является FreeAndNil из SysUtils, которая принимает var-параметр, приводит его к TObject, вызывает ему деструктор, а затем очищает переданный параметр (тоже в виде, приведенном к TObject). Если повсеместно в программе использовался некий класс, а затем он был спрятан за интерфейс, и где-то при var instance: IUnknown забыли убрать вызов FreeAndNil(instance), то это вызов успешно скомпилируется, но эффект его будет непредсказуем.

III. Обертки и агрегаты

Выше говорилось о трудностях, возникающих при попытке обернуть интерфейс без его согласия. С одной стороны, если обертывающий объект пропускает вызовы методов IA через методы своей собственной реализации IA (т.е. осуществляет делегирование интерфейса), то таким способом он сможет делегировать лишь наперед известный набор интерфейсов. А значит, он должен знать, что обернутый экземпляр (пусть это actor) поддерживает именно такой-то набор интерфейсов, и больше никакие. То есть фактически получается обертка над классом, а не интерфейсом, но неявно (уж лучше бы обертка явно была вокруг класса, по крайней мере, тогда была бы проверка этого во время компиляции).

Как же в таком случае работают переходники в COM? Они генерируются автоматически для каждого интерфейса, а если метод интерфейса возвращает другой интерфейс, то вокруг возвращенной им реализации интерфейса тоже генерируется переходник. Это можно рассматривать как возможность для обертки поддерживать неограниченное число заранее неизвестных интерфейсов, добавляя поддержку по требованию. Чтобы реализовать такой подход, нужно, во-первых, иметь возможность генерировать исполняемый код в процессе работы (для чего могут оказаться нужны повышенные привилегии в системе безопасности ОС), и, во-вторых, уметь конструировать обертку, зная лишь идентификатор затребованного интерфейса. Опросить полную информацию об интерфейсе можно, если интерфейс описан в библиотеке типов, зарегистрированной в системе. Если же в вашем приложении не все интерфейсы фигурируют в TLB (ведь для этого придется отказаться в этом интерфейсе от тех типов данных, которые несовместимы с COM: например, обычного string), то непонятно, как агрегирующий объект должен распознавать структуру "недекларированных" интерфейсов.

Начиная с Delphi XE, у программиста есть документированный доступ к внутренней информации о типах (RTTI) выполняемой программы. Это дает бо́льшую свободу, чем TLB, поскольку RTTI генерируется для многих типов, не совместимых с COM. Но все-таки некоторые типы несовместимы и с RTTI; не обязательно что-то экзотическое – например, некоторые типы-перечисления. В принципе, на основе этой информации можно генерировать обертки динамически – как это происходит, например, при использовании CORBA.

Возможно еще одно решение. Если у обертки опрашивают интерфейс, который она не реализует, она может вернуть опросить этот интерфейс у actor и вернуть его. Но это означает, что контроль над ситуацией уходит из обертки, и если теперь возвращенный интерфейс привести к любому другому, будет возвращен интерфейс из actor, уже не обернутый ничем. И кроме того, в этом интерфейсе могли быть какие-то методы, которые обертка, по ее логике, должна была бы обрабатывать как-то особо (для чего и делается перехват – чтобы добавить к каким-то вызовам особую обработку). Более того, "ослабевает" правило симметричности QI: если у интерфейса IA (поддерживаемого оберткой и actor) опросить IB (поддерживаемый только actor), а затем у IB опросить IA, то опрос хотя и пройдет успешно (это требование к QI), но возвращенное значение IA будет отличаться от исходного и вдобавок указывать уже на другой экземпляр. Это не запрещено спецификациями QI, но это ожидание настолько естественно, что в вашем приложении кто-либо из программистов вполне мог написать код, рассчитывающий на полную симметрию (на то, что исходный и полученный указатели на IA совпадают).

Из этой ситуации предложен изящный выход в компонентной объектной модели COM, в ней дается особое определение агрегирования. Оно понимается так, что объект экспортирует, помимо своих интерфейсов, также интерфейсы какого-то другого, агрегированного (того, которым он владеет) объекта. При этом если у агрегированного объекта опрашиваются интерфейсы, поддерживаемые агрегирующим, то он возвращает указатели на интерфейсы агрегирующего. Как нетрудно понять, такое решение (при котором сторонний код даже не осознает, что попеременно работает с различными объектами) требует кооперации обоих объектов, т.е. агрегированный должен знать, что принадлежит агрегирующему, и знать, какому именно. Но если удается выполнить это условие, то такая схема работает.

Впрочем, даже если обозначенная проблема решена, – можно генерировать обертки автоматически, или двухсторонне поддерживается агрегирование – для большинства задач, ассоциирующихся с созданием оберток (мостов, декораторов и т.п.), этого недостаточно. Этого достаточно для знания о факте вызова неизвестного наперед метода, но недостаточно для

дифференцированной обработки этого факта в зависимости от метода: для этого нужно знание о методе. А обертки, как правило, создаются с целью дифференцированной обработки различных вызовов.

В этом состоит одно из преимуществ обертывания классов над обертыванием интерфейсов. Объект, поддерживающий некий интерфейс, может поддерживать и любые другие, и эти другие могут иметь отношение к тем функциям, которые интересуют обертку, но могут быть этой обертке неизвестны (хотя бы на момент ее написания). Конечно, класс тоже может за счет полиморфизма и наследования получить дополнительные элементы поведения, но его поведение при этом не должно изменяться существенно (согласно принципу подстановки Лисков) – если изменяется, то это уже недостаток архитектуры, который можно устранить отдельно.

IV. Неудачное решение: генерализованная система событий и подписки

Допустим, имеется объект, характеризуемый рядом свойств. Свойства представлены другими объектами, агрегированными данным. Свойства могут быть изменены как по запросу их объекта-владельца, так и извне его (например, посредством окна редактирования свойств в стиле «Object inspector»). Необходимо снабдить объект способом реагирования на изменение свойств.

Принцип инверсии зависимостей предлагает уведомлять заинтересованных потребителей не за счет знания о конкретных потребителях, а за счет отсылки их абстрактным приемникам. Для этого есть классический шаблон проектирования "подписчик" (publish-subscribe). Но каков должен быть интерфейс подписки? Прямое и архитектурно экономичное решение может быть следующим. Вводится общий интерфейс IEventProvider, позволяющий подписать слушателя, реализующего интерфейс IEventListener, на событие с обобщенным интерфейсом IEvent. Так, IEventListener может реализовывать метод Notify(event: IEvent). При таком подходе придется одним методом ловить всё, и в нем писать условную логику для различных видов сообщений. И выполнять восходящее приведение типа, чтобы из IEvent получить интерфейс, предоставляющий доступ к информации, специфичной для конкретного сообщения; например, интерфейс IPropertyChangeEvent для сообщения об изменении свойства, или, т.к. свойства бывают разных типов, какой-то другой интерфейс IIntegerPropertyChangeEvent, дающий доступ к типизированному интерфейсу свойства, содержащего целочисленное значение.

Можно оспаривать эффективность именно такого способа представления свойств, но он приведен просто в качестве примера подписки на конкретные типы событий при наличии общего интерфейса в шаблоне «подписчик».

Чтобы избежать восходящего приведения и условной логики, можно под каждый тип событий заводить отдельные интерфейсы источника событий и подписчика; но это дает высокую избыточность системы типов. И все равно при наличии нескольких свойств одного типа придется использовать в методе-обработчике условную логику. В Delphi XE можно использовать обобщенные типы (generics) для обобщенного определения интерфейса отправителя, но с получателями это не сработает.

Как представляется автору, хорошим решением может быть то, которое применяется в VCL или в .Net: ссылки на обработчики событий в виде делегатов. Вместо того, чтобы определять получателя событий как полноценный интерфейс или класс, он может быть определен как ссылка на метод-обработчик, что позволяет определить обработку каждого типа свойства, и даже каждого отдельного свойства, в отдельном методе, причем типизированном, без необходимости восходящего приведения типов. Тип делегата может быть определен как указатель на метод (function of object) или, в XE, как ссылка на функцию (reference to function).

V. ProcessMessages

Визуальная библиотека Delphi предоставляет функцию ProcessMessages, представляющую собой цикл обработки сообщений, продолжающийся до тех пор, пока в очереди событий не закончатся сообщения. Никакой фильтрации сообщений при этом не производится – почти такой же вид имеет главный цикл обработки сообщений в TApplication.

Точное предназначение этой функции неясно. Косвенно выводы сделать можно из ее описания: «In lengthy operations, calling ProcessMessages periodically allows the application to respond to paint and other messages» (в ходе длительных операций периодические вызовы ProcessMessages позволяют приложению отвечать на сообщения о перерисовке и другие сообщения). К сожалению, реализована она так, что ее применение именно с этой целью приводит к двум неприятным побочным эффектам.

Во-первых, если ProcessMessages вызывается, согласно этому рецепту, в процессе выполнения длительной операции, вызванной, скажем, из главного меню приложения, или с помощью «горячих клавиш», то в результате выборки и обработки всех новых поступающих сообщений (а не только, например, сообщений, относящихся к отрисовке окна) пользователь может, пока выборка сообщений продолжается, снова войти в меню и запустить какую-то другую операцию, или воспользоваться другой «горячей клавишей». Вплоть до того, чтобы скомандовать выход из программы – пока запустившая ProcessMessages операция еще не закончилась. Или же может произойти закрытие редактируемого документа в процессе его сохранения, что испортит сохраненный файл. Автор рекомендует воздерживаться от использования ProcessMessages.

Учитывая, что компоненты VCL могут сами вызывать ProcessMessages, один из относительно надежных способов предотвратить проблемы такого рода – блокирование попыток запуска новых операций до окончания текущей. Это несложно сделать, если в приложении есть унифицированный механизм запуска операций (к примеру, к его образованию может приводить следование паттерну «команда», включающему как раз обособление операций). Такой механизм может отследить начало и окончание выполнения операции, или заблокировать запуск.

В некоторых случаях в приложениях используют ProcessMessages, чтобы поглотить сообщения от «мыши», оставшиеся после выполнения запущенной операции. В этих случаях можно воспользоваться функцией PeekMessage с PM_REMOVE для изъятия конкретных групп событий.

Другая проблема более экзотична, но не невероятна. Цикл обработки сообщений в ProcessMessages не имеет ограничений ни по времени работы, ни по количеству обработанных сообщений. Если некий источник будет помещать в очередь новые сообщения с той же (или близкой) частотой, с которой цикл обработки их изымает, цикл может длиться очень долго. В проекте, в котором принимал участие автор, однажды по ошибке один экспериментальный компонент (не несший полезной нагрузки) включал таймер, работавший с частотой 1 мс. Разумеется, реальная частота обработки ограничена квантами времени Windows, но суть не в этом. Если обработка этих событий (на слабых компьютерах) едва успевает за поступлением новых сообщений, а в какой-то момент пользователем запускается длительная операция, содержащая вызов ProcessMessages, то приложение с точки зрения пользователя не завершает операцию, но при этом реагирует на его действия и даже позволяет запускать новые операции. Получается своеобразное зависание, при котором приложение не «висит».

VI. Проблемы отладки

Самое меньшее со времени выхода Delphi 6 и до появления Delphi 2009 отладку реальных программ серьезно осложняла возникающая в IDE ошибка с кодом C0000029. При остановке выполнения программы – на точке прерывания или по команде Pause – в некоторых случаях при попытке перейти на точку исходного кода по ее адресу или по стековому фрейму отладчик показывает не исходный код, а окно дизассемблера, причем это сопровождается непрерывным выводом диалогового окна с невнятным сообщением об ошибке в IDE. Обычно это делает невозможной дальнейшую отладку в этом экземпляре Delphi. Специалисты Embarcadero утверждают, что в Delphi XE проблема устранена, но это не так. Устранен вывод сообщения об ошибке и переход в окно дизассемблера, но отладчик точно так же время от времени перестает работать: реагировать на установку точек прерывания, высчитывать значения выражений и проч., – а значит, нормально отлаживать становится невозможно.

Проблему воспроизвести достаточно просто, создав группу всего из двух проектов общим объемом около 50 строк кода. К сожалению, этот пример является хоть и достаточным, но не необходимым: точный механизм возникновения ошибки автору установить не удалось, как неизвестны и пути обхода. Правда, в Delphi XE проблема возникает существенно реже, и иногда после перекомпиляции (Compile, а не Build!) проектов нормальная работа отладчика восстанавливается.

Еще одна неприятная особенность Delphi в области отладки проявляется при динамическом связывании runtime-библиотеки Delphi. Если для сбора информации об ошибках у конечных пользователей используется какой-либо механизм создания дампа стека, например, такой, о котором рассказывала А. Воробьева из Parallels [3], то авторы программы ожидают, что дамп будет содержать всю цепочку вложенных вызовов функций, приведших к строке, в которой возникла ошибка. Но на деле дамп формируется путем прохода по т.н. стековым фреймам – специальным образом оформленных хранящихся на стеке коротких структур, каждая из которых косвенно ссылается на предыдущую. Эти фреймы создаются самой программой. Delphi предпочитает, опять-таки в целях оптимизации кода, не создавать стековые фреймы, если только не включена директива компиляции Stack frames. А значит, каждая одна из функций, для которых не сгенерированы фреймы, в дампе стека будет "скрывать" информацию о вызвавшей ее функции, в результате чего отладка сильно затрудняется. В создаваемой программе избежать этого легко, достаточно включить опцию создания фреймов. Но вот если используются пакеты времени выполнения (runtime packages), то их тоже необходимо перекомпилировать. А пакет RTL из состава Delphi перекомпилировать непросто.

Этому мешает особый статус модуля System; утверждается, впрочем, что это возможно в Delphi XE2.

Интересным примером последствий этой проблемы является бессмысленность использования функции Win32Check (или OleCheck). Как ей и положено по спецификации, функция реагирует на флаг ошибки и генерирует исключение, но точка возникновения исключения находится как раз внутри самой функции Win32Check (а сама операция, установившая флаг ошибки, напомню, вызывается до вызова Win32Check; оба вызова осуществляются из одного и того же стекового фрейма некой функции F, вызвавшей Win32Check). Происходит следующее: поскольку вызов Win32Check не создает собственный стековый фрейм, то при возникновении исключения последним адресом возврата на стеке в текущем фрейме оказывается адрес внутри Win32Check (а не F), а следующий фрейм принадлежит уже не F, а той функции, которая вызвала F. Потому точку возникновения ошибки можно установить лишь по косвенным признакам.

Такие функции можно перехватить (instrument) функциями со стековым фреймом, и эти перехватчики будут видны в стеке вместо перехваченных функций. Конечно, перехват вызовов функций – в принципе явление не очень распространенное, а в среде Delphi – тем более; но, возможно, существуют готовые решения: как минимум, это появившийся в Delphi XE «штатный» перехватчик вызовов виртуальных методов TVirtualMethodInterceptor. Можно и осуществить перехват самостоятельно, информацию по этой теме найти не так сложно, хотя бы в известной книге Дж. Рихтера [6], раздел “DLL injection and API hooking”.

В Delphi входит довольно удобный отладчик. Он умеет устанавливать точки останова при условии изменения значения по указанному адресу в памяти, умеет выполнять какие-то простые действия при каждом срабатывании точки прерывания, и, что важно, позволяет изменять значения переменных, а также простым способом опрашивать значения строк (для сравнения, в недавних версиях среды разработки Lazarus для компилятора Free Pascal Compiler недостаточно было в отладчике написать просто название строковой переменной, чтобы увидеть текст, содержащийся в строке). Также очень удобна

бывает возможность изменить ход выполнения программы, задав адрес следующей выполняющейся инструкции – разумеется, велик шанс, что после этого отлаживаемая программа будет работать неверно или вообще работать перестанет, но иногда этого хватает, чтобы заново "войти" в только что случайно пропущенную функцию и выяснить, почему она вернула не такой результат, который ожидался. Но просто непостижимо, как можно было с самого момента появления в языке операций динамического приведения типов так и не поддержать эти операции в отладчике! Он просто не воспринимает выражения с ними. Для объектов помогает старый стиль приведения типов: Type(value), но если все полиморфные представления данного объекта как экземпляра какого-либо класса имеют один и тот же базовый адрес (т.е. указатель на любое из таких представлений имеет одно и то же значение), то для интерфейсов это не так! К примеру, если экземпляр поддерживает интерфейсы IA и IB, не связанные отношением наследования, то instance as IA и instance as IB непременно будут иметь разные значения указателя на интерфейс (хотя бы в силу того, что набор методов IB не начинается с методов IA, и наоборот). А значит, если instance объявлен как IA, взятие instance as IB не даст верный указатель на IB, а значит, получить доступ к методам IB через отладчик по-прежнему не получится.

Для таких случаев автор иногда использует простое решение для часто используемых интерфейсов:

```
function GetAsXXX(ref: IUnknown): XXX;
begin
  result := ref as XXX;
end;
```

Казалось бы, в современных версиях Delphi его можно обобщить с использованием обобщенных типов (generics), но компилятор не выводит из описания ограничения «XXX является интерфейсом» возможность использования XXX в качестве типа, к которому выполнять приведение. Возможно, для этих целей вместо as можно использовать механизмы RTTI, ведь проверка типов во время компиляции здесь не нужна.

VII. Управление исходными текстами

Достаточно неприятная ситуация с .res-файлами. С одной стороны, Delphi пересоздает .res-файл проекта при сборке проекта, поэтому добавлять такой файл в систему управления версиями – плохо (файл будет постоянно изменяться у всех участников). Но если не добавлять – придется всю информацию, которая в нем находится, размещать где-то в другом .res-файле, который будет собираться не автоматически, а по какой-то команде. Вероятно, еще более эффективно использовать .rc-файл (исходное задание для сборки .res-файла) и по нему генерировать .res-файл при сборке, при помощи компилятора ресурсов – но сам постоянно меняющийся .res-файл не добавлять. А в .rc-файле подключить и иконку приложения, и информацию о версии, и что-либо еще, что обычно указывается через настройки проекта Delphi.

В Delphi XE правильная обработка .rc-файлов выполняется при помощи директивы компиляции такого вида: {$R 'tt.res' 'C:\MyPath\tt.rc'} в головном файле проекта (.dpr) либо в каком-то из модулей (units). Но компилятор Delphi всё еще несовершенен, и одна лишь эта директива хоть и будет встраивать .res-файл в выходной файл проекта, но не будет вызывать перекомпиляцию .res-файла при изменении .rc. Чтобы обеспечить автоматическую сборку, можно добавить вручную добавить вызов компилятора ресурсов при сборке проекта (это можно сделать через раздел Build Events в настройках проекта).

Вообще-то, есть и гораздо более удобный и стандартный вариант – включить в файл описания проекта (который начиная с Delphi 2007 совместим с MSBuild; еще удобнее бывает передавать MSBuild файл группы проектнов: *.groupproj) элемент, вызывающий обработку .rc-файла компилятором ресурсов при компиляции проекта. Этот способ не является каким-то трюком (будь так, элемент мог бы быть автоматически удален при сохранении проекта), он предусмотрен в IDE и реализуется простым добавлением .rc-файл в проект (в контекстном меню проекта – пункт "Add..."). При этом в .dpr-файл точно так же добавится инструкция $R, но дополнительно в .dproj добавится элемент <RcCompile>, вызывающий компиляцию .rc при сборке проекта.

Следует отметить, что в этой инструкции .res-файл указывается без полного пути к нему. Можно указать и полный путь, но в этом случае следует также указать в настройках проекта выходную папку компилятора ресурсов, т.к. если она не задана, то компиляция из IDE разместит .res-файл «рядом» с .rc-файлом, тогда как MSBuild разместит его в папке файла проектной группы (а искать его затем будет по тому пути, который указан в $R). А раз так, то либо IDE не найдет файл, либо MSBuild.

[1] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, 1st Edition, Addison-Wesley, 1994.

[2] J. Kerievsky, Refactoring to Patterns, Addison-Wesley, 2004.

[3] A. Vorobyova, “Diagnostics experience in virtualization products of Parallels”, in Proc. CEE-SECR 2011.

[4] B. Meyer, Object-Oriented Software Construction, 2nd Edition, Prentice Hall, 2000.

[5] M. Gumerov, “Some Delphi pitfalls”, in RSDN Magazine, 2012, №2, pp. 24-29.

[6] J. Richter, Programming Applications for Microsoft Windows, Microsoft Press, 1999,