Session#: 9022

Title: PANEL: Object Persistence Frameworks

Type: Birds-of-a-Feather

Level: All

Prerequisites: None.

Description: Object Persistence Frameworks (OPFs) are a natural outgrowth of object-oriented design principles, their layered architectures serving to isolate detailed business logic from persistence implementation details in the same way that we've learned to guard it against being embedded in our forms. This session brings together several OPF designers and practitioners for brief explanations of what they are doing and why, followed by a question and answer period on the points or approaches discussed.

Panel discussants:

Joanna Carter, Consultant Software Engineer (www.carterconsulting.org.uk)

Jim Cooper, of Tabdee Ltd. (www.tabdee.ltd.uk)

Bob Dawson, of Integrated DNA Technologies, Inc. (ww.idtdna.com)

Inasmuch as this was an ad-hoc panel discussion, no formal technical paper exists. However, because this was intended to be an all-level, no-prerequisite session, I prepared a set of introductory notes outlining some of the issues and current approaches to Object Persistence Framework design, and a brief bibliography. A revised version of these follows.

An Introduction to OPFs
(Object Persistence Frameworks)

Common OPF Design Goals

Maximize code reuse, maintainability, and reliability by

1. Supporting the creation of Business Objects and persistence code sharable across applications and deployments.
2. Handling common persistence requirements via generic or patterned code.
3. Easing programmer conceptual burden by isolating functional design layers, thereby allowing the programmers to concentrate on single and singly addressable issues.
4. Easing Object Layer/RDBMS coordination problems by binding or decoupling these domains.

Basic Terms and Concepts

Business Object (aka Problem Domain Object or simply Domain Object). Any TObject descendent named after and modeling the relevant properties and methods of an actual object in the problem domain: i.e., TCustomer, TInvoice, TOrder, TAccount, TAddress, etc. The notion of 'actual object' can be quite loose (including what are called 'pure fabrications'), but the general intent is to build up a collection of software objects that collectively mirror the characteristics and behaviors of the real objects, insofar as those qualities are relevant to the programming problem to be solved.

Layered Architecture. The same OO design principle that tells us that one object should not access another s internal implementation also leads us to the conclusion that groups or functional systems of objects should know as little as possible about the inner workings of their neighboring systems, minimizing the degree to which any particular system is bound to, or aware of, the means by which the adjoining systems actually carry out their tasks. An OO programmer would call this functional layering information hiding or loose coupling, but as a design principle it s not confined to OO thinking witness the layered architecture of the internet.

And layered architecture is more than just a good idea in theory it had very practical benefits. For the designer, the separation of programming efforts into discrete layers accurately reflects the fact that each design layer may be constrained or influenced by quite distinct external considerations. For example, the behavior and properties of objects modeled in the problem domain are generally taken from the external real business world of customers and invoices, whereas the choice of persistence platforms might be driven by technological questions, hardware and software acquisition and maintenance costs, existing facilities and staff skills, expected transaction load, etc.) Separation of the program into distinct architectural layers is thus simply a reasonable modeling of the way these issues actually separate themselves in the real world. And for the programmer, the ability to code business logic without constantly interrupting one s train of thought with details of data access and disposition is positively liberating the business of coding becomes more focused, the code cleaner and more maintainable.

In practice, then, a layered architecture design approach usually results in programs that separate into a minimum of three distinct programming efforts:

- Design of the program GUI and user workflow.
- Design of the business objects that the program will manipulate
- Design of a persistence mechanism that the business objects will call on.

Isolating these layers and decoupling their interaction then results in a programming architecture that looks like

GUI

\

Presentation Manager/Abstraction Layer

/

Object Layer (problem domain code)

\

Persistence Manager/Abstraction Layer

/

Database

Object Persistence Framework. An OPF is a core of common code, reusable by multiple applications, that as a minimum provides a highly patterned and consistent way for business objects (BOs) to be called into existence (retrieved from a persistent storage device) and saved (inserted into or updated in the storage device), without allowing the BOs any knowledge of, or dependence on, how persistence is actually being implemented. In application, an object persistence framework generally operates between the program s target problem-domain layer and the persistence layer, in much the same way as the controller component of a Model-View-Controller architecture mediates between the business object layer and the GUI, isolating the former from the specifics of the interface design. In each case, the aim is to isolate the core business logic or problem domain objects from the essentially unrelated technical details of the interface design in front of it, or the persistent storage behind it.

Principle of BO Identity. Any persistent instance of a business object may be assumed to represent a unique and uniquely identifiable instance of the problem domain class being modeled and is persistent (savable/recallable) by that identity. This is sometimes referred to as the Object Identity (OID) design pattern. Object Identities are routinely (although not necessarily) used as database primary keys for their respective tables, and so share the same characteristics: uniqueness, immutability, and non-representationality.

Alternative Approaches to OPF design

If we ask "What is the best way to separate BO properties and behavior from business object persistence?" then we're going to encounter some widely divergent answers. The following is a short catalog of general approaches:

Naïve RTTI-based approach. Derive your business object from a class designed to handle persistence in a methodical, generic way, using the generic self-inspection

capabilities provided by TPersistent. An abstract business object ancestor is built to interrogate/iterate its published properties and, based on what it finds, build load/save code dynamically. For example, an object would build an SQL update statement to hit a table named whatever its classname is, then iterate its own published properties to generate the field names and current values, and finally append a where clause of

```
'WHERE OBJ_ID = ' + IDToStr(self.ID).
```

The persistence code for such an object really consists of the base object s use of the TypInfo unit as a means of serializing its properties and determining their data types it really doesn t matter whether it uses what it finds to build SQL statements, or simply to read and write its values to an INI file (or to a .dfm file).

Upside: Probably the smallest possible codebase, because everything is essentially being done in an entirely generic manner by the ancestor without regard for what published properties specific descendents actually have. Very useful for proofing quick proof-of-concept models.

Downside: Major data containment problems (to be persisted, a property must be published), and serious BO/database design coupling and efficiency issues.

Dataset/Datamodule based approach. Base your business object on a class that Delphi already provides for you for handling persistence issues, the TDataModule. The program on the front end uses the business object exposed by the datamodule but does not ever access the persistence internals directly. Alternatively, derive a custom TDataSet descendent that understands its own rows as representing the properties of a specific object class.

Upside: this is a cake-and-eat-it-too approach if you're a fan of data-aware controls, because if you do it right you can expose business object properties to DA controls in your front end as TFields.

Downside: Descending from a data module or data-oriented object means that while you may provide OO methods and properties, you can t really achieve OO containment it may be too easy and too tempting to bypass the object structure.

Class-pairing/linking framework approach. Code business objects without any knowledge at all of persistence, other than the ability to throw themselves at another object to ask for help. Descend persistence helper classes from an ancestor class that any business object can call, then use a persistence framework to broker descendent business object and persistence-helper connections, so that a business object's internal save method might look like this:

```
begin
 if not assigned(FPersistenceHelper) then
  FPersistenceHelper := FFramework.GetPersistenceHelperFor(self.classname);
 FPersistenceHelper.Save(self);
end;
```

Upside: about as pure a separation as you can get between the business object and the persistence helper classes while still allowing direct data passage, with a high resulting freedom to alter either layer as required without damaging or impacting the other. Additionally, the basic concepts and example code for a simple implementation of this approach are easily available through Borland s CodeCentral repository, courtesy of Philip Brown (see bibliography).

Downside: Since each business object requires a persistor/partner, this approach can result in a lot of repetitious code devoted to information passing between the business objects and their associated persistence helper classes.

Persistence-mapping approach. Provide in your persistence layer a set of classes that can map a business object to its persistence statements, as mediated by the current persistence connection. This can be done either dynamically (the persistence layer generating SQL statements on-the-fly as required) or statically (the persistence layer being dependent on some form of data dictionary from which it can retrieve instructions on how to proceed. The basic concepts of this approach (with notional class diagrams) are discussed by Scott Ambler (see bibliography).

Upside: Dynamic mapping can provide an extremely small, reliable code base. Static mapping can accommodate legacy database designs.

Downside: Dynamic mapping may result in relatively inefficient SQL, and may result in tight coupling/constraints between BO class design and database design. Static mapping introduces the need to maintain explicit metadata (mapping tables/data dictionaries).

Streaming/TransportClass/XML-based approach. Business objects are required to implement (either as inherited methods or as interfaces) the abilities to read themselves from or write themselves to a single transmittable format. This format may be implemented as an XML structure, a stream, an adapter/boundary class, or any other specific data-porting mechanism that occurs to you. The approach might be coupled or contained within a visitor/visited pattern as well, so that saving a collection of objects would be accomplished simply by iterating a visitor across each object to be acted on. The primary hallmark of this approach is that that data passage is now totally blind: a BO is no longer bound to any specific persistor class or method at all (often, the structure at the persistence-layer end of the pipe in this approach is an engine designed to carry out mapping functions, as described in the previous approach).

Upside: Considerable opportunities for generic/polymorphic code. Division of problem into data transport and persistence actions maximizes flexibility.

Downside: Introduction of the transport mechanism as a distinct layer makes this more complicated to implement than other approaches.

Caveat: the general approaches here are not necessarily mutually exclusive alternatives technique mix and match is a possibility at various points.

Questions to ask of OPF implementations

Creating a persistence framework could be as simple as writing a single class that knows how save any descendent object, or as complex and capable as the VCL itself. Here are a few issues to keep in mind as you evaluate designs or sketch out your own:

1. How does the OPF accomplish its data transfer tasks without breaking OO data containment?
2. Does the OPF demand that all BOs descend from a specific class, or can any object be persisted by implementing a specific method or interface on it?
3. How does the OPF provide for BO collections or lists?
4. How does the OPF address BO-BO interaction and navigation?
5. How does the OPF provide for persistence transactions?
6. Does the OPF accommodate BO instance and collection exposure on the GUI via data-aware controls?
7. How flexible is the OPF regarding web/multi-tier implementations?
8. Can the OPF accommodate legacy database structures, or does it require green-field design?
9. How, or does, the OPF address issues of BO instance contention when interacting with the persistence storage? (Either within a fat client or across multiple thin clients.)
10. How, or does, the OPF address issues of BO instance contention when interacting with the GUI?
11. How fast/efficient is the persistence layer?
12. What ancillary services, if any, does the OPF framework provide?

Further reading

(no attempt to be comprehensive, just some easily available things we've found helpful)

Introductory tutorial:

Brown, Philip. An Object Oriented Persistence Layer Design. Philip published a series of introductory articles in the late EXE magazine (UK). These are available from the Borland Community CodeCentral website, in the category Best Techniques / Delphi, submission #15511.

Design discussions and ideas:

The borland.public.dephi.oodesign forum is frequented by all the session panelists, and by many other Delphi programmers interested in object-oriented design in general or OPF architecture in specific. Frequent discussions of OPF issues and approaches can be found there, and you can generally always get answers to your questions (possibly even multiple conflicting answers).

Ambler, Scott. http://www.ambysoft.com/ Has many interesting pieces in the online white papers section, including a paper containing a general class model for a complete persistence layer: Design of a Robust Persistence Layer for Relational Databases.

Carter, Joanna. http://www.carterconsulting.org.uk See her four-part Keeping Hold of Your Things article, and the link to A Series of Articles on Object Persistence.

Fowler, Martin. http://www.martinfowler.com/. From his main site, use the Articles link at the top of the page. From the articles page see the pieces Separating User Interface Code and Reducing Coupling. Martin has removed a lot of the enterprise architecture material from his website since its inclusion in his new book, Patterns of Enterprise Application Architecture (ISBN: 0321127420). Recommended.

Projects

www.borland.com

ECO/Bold: A commercial model-driven, layered-architecture application framework and model-driven development tool (calling ECO or Bold a persistence framework is a bit like calling a car wheels not inaccurate, but incomplete).

http://sourceforge.net/projects/obiwan/

The JEDI Obiwan project's main objective is to provide an object oriented specification and API for access to persistent data using Borland Delphi and related environments/

languages.

http://www.techinsite.com.au

The TechInsite Object Persistence Framework (tiOPF) is an Open Source framework of Delphi code that simplifies the mapping of an object oriented business model into a relational database. The framework is mature and robust. It has been in use on production sites for over five years. It is free, open source, and available for immediate download with full source code.

www.seleqt.com

A former commercial persistence framework (rumored to be going open source):

https://sourceforge.net/projects/depo

Delphi Persistent Object (DePO) is a Object Persistent Framework. The implementation follows strictly the Scott Ambler implementation, adapting itself into the borland VCL.

And what about .NET?

In addition to Borland s Enterprise Core Objects (ECO), a .NET-specific redesign of Bold, there are a growing number of both commercial and open source OPF packages available for the .NET developer. Here are a few we re aware of (text blurbs from the respective websites):

http://www.deklarit.com/

DeKlarit provides a simple way to automate a great part of the application design, development and maintenance process by automatically generating the data model and the code for your data access and business logic layers.

http://www.norpheme.com/

Norpheme is an object-relational persistence framework for .NET. Unlike code generators, Norpheme doesn't generate thousands of lines of code, it eliminates thousands of lines of code. Norpheme enables reuse and ensures a layered application architecture. It accelerates projects while lessening total lifetime costs.

Norpheme leverages the power of ADO .NET and at the same time insulates application developers from the arcane complexities of the ADO .NET data access API and vendor-specific data clients. Norpheme s object-relational mapping allows developers to build business-oriented middle tier components that more naturally describe relational data in the object-oriented .NET environment.

http://www.objectpersistence.com/

Persistence.Net provides the missing link between your object model and relational data store. With the simplicity of .Net attribute-based programming, you will find adding database persistence to your application is as easy as [Pie(filling= apple)].

http://www.codeproject.com/csharp/opfnet.asp

The Object Persistence Framework for .Net (OPF.Net) is a complete set of classes that implement an object-relational mapping strategy for object oriented access to traditional relational database management systems and other types of persistent storage types such as XML files. OPF.Net has been designed and implemented for practical use in small to medium size projects and is currently being successfully used in several projects.

http://sourceforge.net/projects/sisyphuspf/

A object persistence framework for .NET implemented in C#. Provides a simple way to persist objects that allows developers to hook in custom validation and business logic.