Container Based Persistence

Brian A. Berenbach
Siemens Corporate Research, Inc.
755 College Rd. East
Princeton, NJ 08540-6632
Tel. (609) 734-3395
brian.berenbach@scr.siemens.com

ABSTRACT

This position paper describes an approach to managing persistence and events with object-oriented three-tiered architectures. It has the advantages of simplicity, portability and extensibility. The architecture has been used successfully on several mission-critical projects. It has the added advantage of enabling application developers to use a container-based API to retrieve and store objects with no knowledge of the underlying storage mechanisms. It also simplifies event management where real-time client notification of changes to persistent data is required.

INTRODUCTION

Several projects that I worked on recently appeared to have the same set of requirements. They were:

- Object persistence to a relational database
- Multiple clients (users) supported by a single server
- A requirement to notify client objects whenever something they are viewing or manipulating has changed in the database
- The need for less experienced developers to extract, manipulate, and store objects with no knowledge of the underlying persistence mechanisms
- Coherent persistence of complex aggregate objects

This position paper describes a simple solution for meeting such requirements that I have used recently on three projects, one using Visual Basic and two with Java. I will focus here on the concepts and generic methods rather than on the implementation specific details. While there are opportunities to use or combine existing design patterns with this approach (e.g. Composite, Proxy, Chain of Responsibility), I have avoided using them in order to clarify the relationships between the several simple classes defined here and the primary mechanisms used. Depending on the domain in which the application is used, some of the well-known patterns such as those found in Gamma[1], et. al. may or may not be appropriate. I have used the Proxy pattern[1] in the example shown for container access to the database and both container and client access to the event server. Depending on the implementation backbone, a server proxy may not be necessary. For example, with DCOM it would be required, as events cannot fire across nodes; with other communication backbones such as CORBA, an event proxy would not be necessary.

Developers on projects with an object-oriented architecture often spend an inordinate amount of time developing code to support persistence with relational databases. When the objects start to get complicated, for example, multiple levels of aggregation, the code can be ad hoc and complicated. This results in hard-to-maintain software that is difficult to modify if a redesign of any kind is required.

Another problem that developers sometimes have to grapple with is that of multiple, possibly distributed views of objects. For example, a payroll manager brings up a view of an employee, and then brings up several ancillary screens all related to the same employee, some containing the same data. A change is made to some data, and it is expected that the other view(s) on the same monitor, and on any other payroll or human resource computers will instantly reflect the change.

Not all developers should be able to (or want to) grapple with persistence when using existing software building block classes to assemble complex systems. A uniform persistence mechanism transparent to application developers can improve both productivity and reliability.

The set of cooperating classes shown in Figure 3 provide a simple set of mechanisms for designing three-tiered client-server systems that provide solutions to the above issues. My design is specifically targeted at "groupware" solutions; e.g., software solutions for teams or groups of about 50 concurrent users. I have not tried it on larger applications. I would also recommend benchmarking before attempting to use the proto-pattern described here.

Figure 1 illustrates proto-pattern in action. A form used for editing an object uses the object's container to retrieve the object from the database. The object is edited through the form, and the form then requests the container to save the object. The container executes the object's save mechanism, and upon the transaction completing successfully, raises an event through the event server. The server communicates with any active event proxy to notify it that the specific object has changed in the database. Any active form (including the one originally requesting the change) is notified by the event proxy that the object has changed in the database.

[1] Gamma, E., Helm, R. Johnson, R., Vlissides, J. *Design Patterns.* Addison-Wesley, Reading, MA. 1995

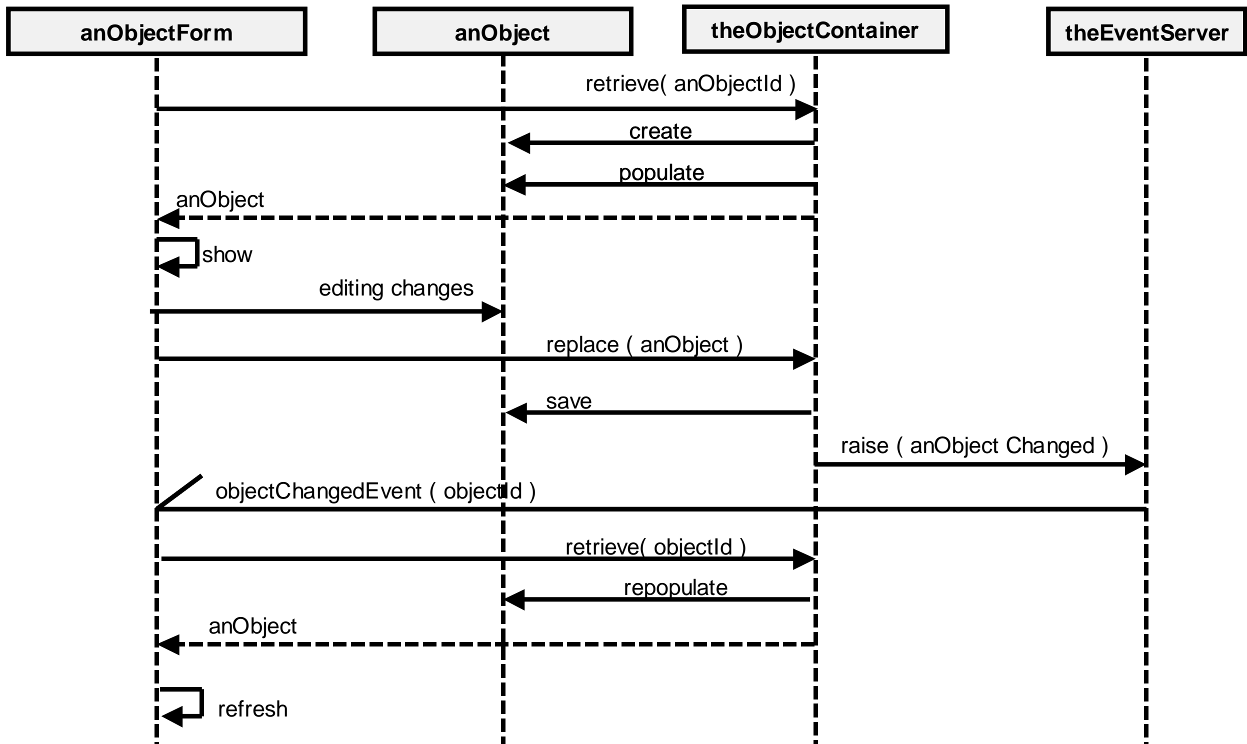

Figure 1 Sample Process using Container Based Persistence

The object id is extracted from the event, and if relevant, the form refreshes itself by throwing away the existing object and pulling a new copy from the database, then refreshing its view(s).

Depending on the specific software application, the views may not require a refresh. However, this is a very clean way of handling situations where a single client may have multiple windows up viewing related information, or where the user is looking at multiple views of the same object on different client machines simultaneously (a control system, for example).

THE CONTAINER

Customized containers that inherit from the base Container class shown here is the cornerstone of container based persistence. Methods are provided to manage persistence and hide any complexity from the application developer. Containers for specific types of objects can have additional methods that perform checks and/or other functions. An added plus is that the container can be considered a "fat" business object and can be used by other applications that need access to the objects that the container stores.

The Container base class methods are described in the table below.

Method	Description
add	This method adds the item to the container, and persists it to the database. If the object already exists in the database an error will be returned and no action will be taken.
check	This method checks contained objects for completeness according to specified business rules. If no object is specified by name or id, all the objects in the database are checked. A container of Ids is returned. If all the contained objects were ok, the container will be returned empty. If any objects were not complete, a container of those object Ids is returned; e.g. so that it can be displayed in a list box.
copy	This method is used to perform a "deep" copy of the internals of an object to a new object. All the properties, subject to the object type business rules, are copied to the new object. An error code is returned. The new object may not be in the database. If the receiving object already is in the container(i.e. persistent), an error will result.
delete	This method deletes the item from both persistent store and from the container if it is currently in memory. The item is specified either by name or by id.
exists	This method returns a Boolean yes/no As to whether or not the specified object exists in the container. The item can be looked up either by name or by id.
export	This method exports all its contained objects to the specified file As an XML "dump". If the file already exists the contained items are appended to the existing file.

Method	Description
IdToName	This method converts an object Id to an object name and returns the object name. It is typically used when displaying objects in a list box.
isEmpty	This method returns a Boolean **True** if the container is empty and false if it contains objects.
nameToID	Given an object name as a string, this method returns the object id. Example: a user drags the name of a customer (copies) from a customer list box into an order list box. The name is converted to an id, and the id is then stored in the customer order object.
rename	Renames an object both in memory and in the database.
replace	This method replaces an existing object with the new version passed. If the object does not already exist in the database an error will occur. This method is typically used where an object is retrieved from storage, changes are made through the GUI, and then the user saves the changes. Note that the replacement cascades. For example, if an employee has deductions, when the employee object is saved, any changes to the employee's deduction objects will also be saved.
retrieve	Given an object name as a string or the object, this method determines if the object already exists in memory. If it does, a pointer to the object is returned. If it does not, then it creates an object, retrieves the object content from the database, populates the object, and returns a pointer to the object. If the object does not exist, a null pointer is returned.

OBJECT PERSISTENCE AND DATA BASE STORAGE AND RETRIEVAL

Data base storage and retrieval is accomplished with a proxy object using either ODBC or JDBC. When an object is stored in the database for the first time, a unique id is assigned by the database. This numeric key is then used when performing database operations on the object and any of its aggregate components.

Use Of Transactions

Transactions are used to insure that the entire object and any of its aggregate components are saved together, and that "dirty" or partially updated objects do not exist in the database.

Object Id And The Primary Key

Every persisted object has a unique id property (**Id**). This id will be null when the object is first created, and after it has been saved to the database it will have a numeric value, which is the primary key value for the table that it is stored in. The save method of the object looks at the id property to determine whether to perform an INSERT (new item) or UPDATE (replace existing item) operation.

Aggregated Objects and The parentId Property

When an aggregated object is created (this is an object which does not exist by itself, but is a part of another object) it is passed the id of the parent object of which it is an aggregate and it is stored in the **Id** property. When the object is saved to the database, it then uses the **parentId** property as a foreign key (Figure 2) to establish a relationship between the parent (whole) and aggregated (is a part of) components. Using the example of an employee who has payroll deductions the entity relationship would be expressed as:

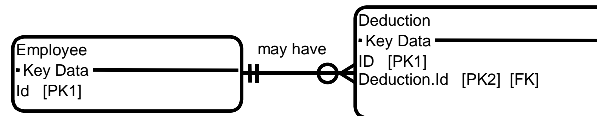

Figure 2 Aggregate Object Persistence

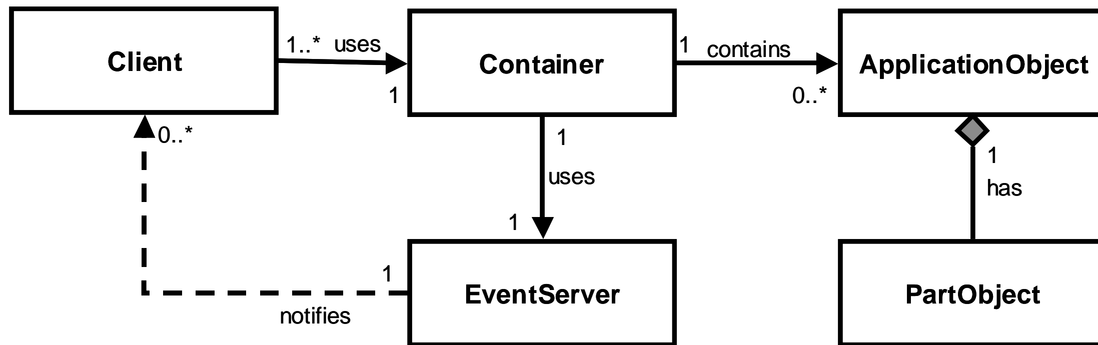

Figure 3 Participating Classes

Sample Save Operation

For example, here is the set of steps that occur when an employee object is created and saved, and then a new deduction is added and the employee is saved again.

1. A new, empty employee object is created when a "new" employee object form is activated.
2. The form is filled in.
3. A deduction object is added to the employee object's deduction collection. An error message occurs because you can't add a deduction to the employee until the employee object has been saved to the database (see step 9).
4. The "save" button is pressed. The employee object form requests the employee object container to save the employee object.
5. The employee object container executes the save method of the employee.
6. The employee is saved to disk as an "**INSERT**" operation inside a transaction, and the unique id of the new saved employee is returned and stored in the employee object in memory in the **Id** property.
7. The user then adds a deduction. When the deduction object is created, it is passed the employee **Id** value that is stored in its **parentId** property (in this example Employee.Id). It now knows which employee it is a part of (aggregation).
8. The employee is saved using the replace method: **EmployeeContainer.replace(anEmployee)**
9. In the save method, a transaction is initiated (**BEGINTRANS**), an **UPDATE** is done since the employee already exists in the database, and the aggregate components of the employee object are iteratively saved by sending them messages to execute their save methods.
10. Inside the deduction save method, it notices that the **parentId** property has no value, i.e. it has not yet been saved to the database, so an **INSERT** is done to save the deduction. If the deduction had already been saved (i.e. the deduction property **aDeduction.Id** has a valid value) then an **UPDATE** operation is done instead of an **INSERT** operation. The compound key used to access the deduction will be "<employee.Id>.<deduction.Id>".
11. Since everything was saved successfully, the save operation is completed with a **COMMIT** and an error code of success is passed back to the original requestor. If any failure had been detected, a **ROLLBACK** would have been executed, and the appropriate error message would have been passed up to the client requesting the replace service of the container.
12. If the operation completes successfully, an event is generated to notify other desktop clients (and other forms in this desktop that may be looking at the object) that a change has occurred. Whenever an object and its contained objects and properties are persisted to the database, the operation is conducted as a single atomic transaction.

EVENT HANDLING

Depending on the implementation environment, a Proxy for handling events may be required. DCOM, for example, does not allow raising events across nodes, so a proxy residing on each local client machine raises events to notify any processes running on that client that an event has occurred. Any forms or processes that need to be aware of data changes then can listen for events. For example:

- When an event occurs each form is notified
- The form checks the context of the event to see if the objects being viewed and/or manipulated are related to the changed item(s)
- If they are related, the form does a refresh by discarding the currently viewed objects and going back to the container and getting a new copy of the relevant object(s)

CONCLUSIONS

Container based persistence was found to be an effective architecture for "groupware" three-tiered client-server applications. The productivity of application software developers was high, as they did not have to spend time implementing persistence mechanisms. The architecture was found to be robust and scalable, and had the added benefit that other applications were able to connect to and use both the container and event handling mechanisms. This architectural approach also eliminates the "stovepipe" restriction commonly seen on client-server applications.