Technical Note

An Example of Code Structuring With Rep++

Publication date: January 2012

An Example of Code Structuring With Rep++

Overview

This article identifies some of the issues that programmers face when developing multi-tiered systems, and proposes an approach for structuring your code that simplifies overall code development and management.

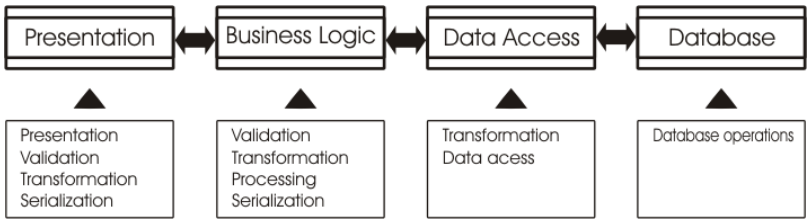
Context

Architecture of information systems

There are several ways to define the architecture of an information system. Generally, we tend to implement architectures with four or even more tiers: the more tiers there are, the more versatile and reusable a system becomes. However, this comes at the cost of an increase in quantity and complexity of code, mainly because of:

- communication between tiers
- duplication of code and structures in different tiers.

Nowadays, the majority of implementations use an object-oriented design to develop the tiers, and objects are used to represent the different entities on each tier. The example below depicts a four-tiered system:



- **Presentation**: where users interact with the system. Common functions include validation, transformation, presentation and serialization of the data.
- **Business logic**: responsible for data processing, this tier implements business rules. Common functions include data validation, transformation, processing, serialization.
- **Data access**: manages the relation with databases, Web Services, CICS, BizTalk, etc. Common functions include transformation and database access.
- **Database**: the database itself.

In the system above, business objects are present across the tiers; some of their functions are used in several tiers, while others are specific to a tier.

Don't Repeat Yourself : The DRY principle

An important concept in programming is the DRY principle. The DRY principle[1] puts forward the idea that any piece of knowledge (code, definition, schema, etc.) should have a single, unambiguous authoritative representation in an information system. Following this approach reduces duplication of logic and code, which are sources of errors and ambiguities, thereby facilitating application maintenance.

Unfortunately, with the multi-tiered systems that are developed today, it becomes increasingly difficult to abide by this principle since an entity will need to be defined at every tier.

The goal of this article is thus to describe a way to implement a DRY approach to develop multi-tiered systems while greatly reducing or eliminating the duplication that often occurs.

Metadata, meta-objects and typed objects

In a traditional object-oriented project, objects represent specific entities, with a fixed set of properties. The objects carry the data while the definitions are hardwired in the code. This decreases the flexibility of the system and increases the duplication of the definitions in different objects and in different tiers.

Rep++ facilitates the development of a DRY information system because it centralizes a system's definitions, i.e., the metadata, in the repository instead of hard wiring them in each object. This approach makes the metadata shareable and always synchronized across:

- the different tiers
- the different objects in each tier.

For instance, if we define a CLIENT object, the same definition will be used in the different tiers. In the same way, if we define a field called CLIENTCODE, the definition will be shared by all objects that include this field. If the definition changes, then all objects that include the field will automatically use the new definition.

To implement the objects found in the different tiers, Rep++ uses meta-objects and typed objects. Meta-objects are generic objects whose interface is independent from the objects they represent. Their definitions (or metadata), stored in the Rep++ repository, provide you with a series of base functions such as unit validation, presentation, transformation, and serialization. Through reflection, meta-objects give you access to the metadata (and to the base functions). Should the metadata change, the meta-objects would automatically adapt.

Typed objects are derived from meta-objects. They inherit all the advantages of the meta-objects (reflection capability, generic character), and also support IntelliSense®. Typed objects are used to contain your code.

Once the meta-objects are defined and the typed objects derived from them, all there is left to do is to manage your code, i.e. the code that is not handled by the base functions of the meta-objects. Sometimes this code is specific to one tier, sometimes it may be used on several tiers; sometimes it applies to a single object, sometimes to several. The big question is: How can you structure this code while respecting the DRY principle?

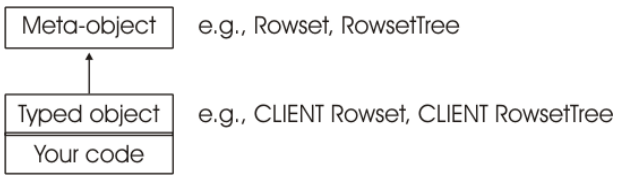


Figure 1. The definition of a meta-object such as a Rowset or a RowsetTree is stored in the repository. This definition is available to all typed objects that derive from the meta-object.

[1] The Dry principle was proposed by D. Thomas and A. Hunt in their book "The Pragmatic Programmer" (Addison-Wesley, 1999).

Strategies for structuring your code with Rep++

Avoiding duplication

Rep++ solves the duplication issue for the declarative part of the code by storing all the definitions in its repository, but the rest of the implementation code must be distributed between objects and tiers in a way that avoids replication.

Making sure your code is called at the appropriate time

It is always a good idea to apply the DRY principle and have a single definition of the elements of your system, but it is often insufficient. Let's say you want to perform a special validation on a particular field. The obvious solution is to call this validation manually everywhere the field is used in your program, but this can be a tedious and error-prone task. A more efficient solution would be to associate the validation to the target field, find a single location for the validation code, and make sure it will be called automatically.

JIT call

A JIT (Just In Time) call is an implementation approach that ensures that a procedure, a function, a validation, etc., is invoked at the right place, at the right time, and only when necessary, without any explicit intervention.

JIT calls can be used to ensure your validation code is called at the right moment. Two implementation approaches are described here, according to whether your code is linked to business objects or not.

Code linked to business objects

For code linked to business objects, such as Rowsets and RowsetTrees, a series of "placeholder" methods in the meta-objects and typed objects are especially designed to store your custom code. Most of these methods are triggered by Rep++ events related to database or validation operations, while the rest are called by a framework through the IRowsetCustomCode and IRowsetTreeCustomCode interfaces.

Method	Description
TypedRowsetBase.CCDbAfterDelLine	Allows the programmer to insert additional code after a line is deleted from the database.
TypedRowsetBase.CCDbAfterInsLine	Allows the programmer to insert additional code after a line is inserted in the database.
TypedRowsetBase.CCDbAfterReadLine	Allows the programmer to insert additional code after a line is read from the database.
TypedRowsetBase.CCDbAfterUpdLine	Allows the programmer to insert additional code after a line is updated in the database.
TypedRowsetBase.CCDbBeforeDelLine	Allows the programmer to insert additional code before a line is deleted from the database.
TypedRowsetBase.CCDbBeforeInsLine	Allows the programmer to insert additional code before a line is inserted in the database.
TypedRowsetBase.CCDbBeforeReadLine	Allows the programmer to insert additional code before a line is read from the database.

TypedRowsetBase.CCDbBeforeUpdLine	Allows the programmer to insert additional code before a line is updated in the database.
TypedRowsetBase.CCFldAfterValidAns	Allows the programmer to insert additional validation code or to perform extra processing after a successful field validation.
TypedRowsetBase.CCFldBeforeValidAns	Allows the programmer to perform some additional processing just before validation is performed.
***TypedRowsetBase.CCRowsetAfterLineOperation**	Allows the programmer to perform some additional processing just after a line is added or deleted.
TypedRowsetBase.CCRowsetAfterValidLine	Allows the programmer to perform some additional processing just after a line is successfully validated.
***TypedRowsetBase.CCRowsetBeforeLineOperation**	Allows the programmer to perform some additional processing just before a line is added or deleted.
TypedRowsetBase.CCRowsetBeforeValidLine	Allows the programmer to perform some additional processing just before a line is validated.
***TypedRowsetBase.CCRowsetValidate**	Allows the programmer to perform additional validation on the Rowset object as a whole.
TypedRowsetTreeBase.CCDbAfterReadRowsetTree	Allows the programmer to insert additional code after a rowsettree is read from the database.
TypedRowsetTreeBase.CCDbAfterUpdRowsetTree	Allows the programmer to insert additional code after a rowset tree is updated in the database.
TypedRowsetTreeBase.CCDbBeforeReadRowsetTree	Allows the programmer to insert additional code before a rowset tree is read from the database.
TypedRowsetTreeBase.CCDbBeforeUpdRowsetTree	Allows the programmer to insert additional code before a rowset tree is updated in the database.
****TypedRowsetTreeBase.CCRowsetTreeValidate**	Allows the programmer to perform additional validation on the RowsetTree object as a whole.

* indicates the method implements the IRowsetCustomCode interface.
** indicates the method implements the IRowsetTreeCustomCode interface.

Code not linked to business objects

Some of the programmer code is not strictly associated to business objects such as Rowsets and RowsetTrees, but instead to other objects such as fields, which may be contained in many different business objects. For instance, you might want to perform a special validation for the CLIENTCODE field. If you use the placeholder methods described above, like TypedRowsetBase.CCFldAfterValidAns, then you will have to repeat the validation call in each object that includes the field, defeating the purpose of JIT calls.

To implement the JIT approach in this case, you can use an observer pattern.

Observer pattern

An observer pattern is a pattern where an object (the observer) "listens" to events and steps in only in particular instances to call routines to perform additional tasks.

To define the observer pattern, you can intercept a Rep++ event, such as Application.FldAfterValidAns, to call a routine every time a field is validated. The observer must listen to this event at all times and for all fields: it will look for the fields of interest (CLIENTCODE in this case, but there may be many) and call the special validation code accordingly. The observer can filter the fields based on:

- the field name
- the field type
- a user-defined attribute
- etc.

You can use the observer pattern on field events, but also on events related to Rowsets and RowsetTrees. Together, JIT calls and observer patterns can be used in many ways, such as:

- To log transactions for all marked RowsetTrees
- To define your own field validation
- To encrypt/decrypt targeted fields
- etc.

Taking advantage of inheritance

There are a few strategies to control code duplication between tiers.

For instance, a CLIENT object can contain code that applies only for the presentation layer, such as how to represent negative values (in red, between parentheses or with negative sign), while some other code could be useful in both the presentation and the business logic tiers. On the one hand, you could have a single object that contains code that applies to all tiers. This approach avoids duplication, however the object would need to carry a lot of code that is useless in one tier or another. This is a suitable approach, but the trade-off might not be acceptable in your context.

On the other hand, you could have many objects that only carry the necessary code for each tier. This second approach implies lots of duplication since some of the object's code must be repeated for each tier.

To avoid code duplication and keep your code DRY, one solution is to use inheritance. In Figure 2~~Figure 2~~, the base CLIENT object (A) contains the code used by more than one tier. CLIENT objects (B) and (C) inherit from (A): CLIENT (B) contains the code specific to the presentation tier, while CLIENT (C) contains the code specific to the business logic tier. This solution prevents useless redundancy, while the objects contain just enough information to carry their tasks.

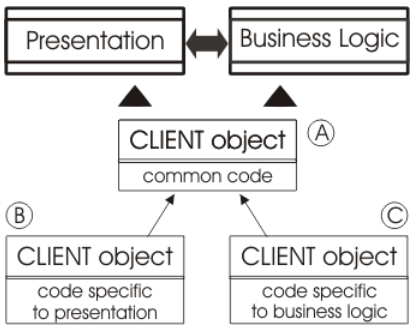


Figure 2. Sometimes your added code can be specific to one layer, as in (B) and (C), sometimes to several layers, as in (A).

Managing errors

The routines that perform validation do not know how to display errors, because they can originate in user code, Rep++ code, Windows or web clients, on different tiers, processes or computers. In all cases, we need a consistent way to return errors, where the calling module can retrieve them. The devised approach is for the routines to return errors in an **Errors** collection. The Error objects of the collection provide information about the error: number, level, message, object type and a reference to the object that triggered the error. The Errors collection is not directly serializable because the Error objects refer to complex objects that are not serializable. On interprocess or inter-computer calls, however, serialization is necessary for the errors to be transmitted or transferred between processes or systems.

The Rep++ toolkit includes a class, SerializableError, which is a serializable version of an Errors collection. Its goal is:

- To serialize errors so that they can be communicated effectively between tiers, technologies, processes, machines, etc.
- To store errors from more than one routine call. When an error is generated following a method call, it is stored in the Errors collection. However, when a new call is made, the collection is first emptied so that its content reflects only the result of the current call. By using the SerializableError object, errors from different calls can be collected and preserved, thereby controlling their lifetime.

The SerializableError class therefore helps you manage errors, regardless of where they were generated, so that you can deal with them in an appropriate manner.