re-store

reader requirements

This page assumes that you have understood

- the overview of this section
- the concept of domain objects (when in doubt, read wikipedia
- basic concepts of O/R-mapping

re-store persists domain objects

re-store is re-motion's O/R-mapper, but also responsible for transactions. re-store persists *domain objects* in an arbitrary database, but at the time of this writing, only Microsoft SQL 2005/2008 is supported. Writing an adapter for other databases is an advanced programme, but thanks to good Linq-integration ("re-linq") and well thought-out interfaces a straightforward affair of software engineering. Such integration is not limited to relational databases, by the way. Object-oriented databases or XML-stores are also welcome with re-store.

(If you need a backgrounder on O/R-mapping in general: Persisting objects in tables.)

re-store supports

- single table inheritance
- concrete table inheritance

At the time of this writing, re-store does not support class table inheritance. The good news is that you can build good O/R-mapping from a combination of single and concrete table inheritance. (Fairly embellished plans exist, but don't hold your breath.) For learning how to control single and concrete table inheritance, read the corresponding section of the PhoneBook walk-through, the The DBTable attribute (Location class).

re-store can do its work - mapping from properties in .NET classes to columns in tables - after a corresponding schema has been derived by program generation. This is done by a command-line tool named dbschema.exe.

re-store does lazy loading

Persisting objects is not worth anything without re-loading persisted objects when they are needed. re-store supports "lazy loading", i.e. only domain objects needed for a given operation of the user are actually loaded into memory. Domain objects are loaded in their entirety. If a domain object references other domain objects it is typical that all referenced domain objects get also loaded, because their display name is stored in them and is needed for completely displaying the referencing domain object. Since re-store does not load single properties, all those reference domain objects are loaded.

re-store manages *client transactions*

re-store provides a powerful transaction system. A *client* transaction, as re-store transactions are called, can spawn sub-transactions and limit them to their own *transaction scope*. You can build arbitrarily complex trees of transactions and sub-transactions. A *commit* in a sub-transaction writes modifications back into the *parent* transaction that spawned the sub-transaction. In this fashion, modifications bubble up from sub-transaction to the *root* transaction. As soon as the root transaction is committed, re-store actually writes the accumulated modifications into the database. This is important for typical re-motion applications where users can open child forms from parent forms by following a reference property to a referenced domain object with its own form. If the user decides to discard modifications at any one point in the tree of transactions/opened windows, then the modifications of all children are forgotten and don't make it into the database. The ultimate example is pressing the *Cancel* button in a form that spawned a bushy tree of child forms/transactions, because this window corresponds to the *root* transaction. Canceling the form means discarding all modifications that have accumulated in that root transaction.

Transaction scopes are a programming mechanism for keeping track of where in the tree of transactions execution of the program is located. Transaction scopes keep variables containing (sub-)transaction objects local to the parent transaction and give the programmer a natural sense on which part of the code belongs to which transaction.

Client transactions and transaction scopes are discussed in detail on these pages (copied from the PhoneBook tutorial).

re-store also persists relations between domain objects

With re-store, you can declare reference properties to have

- unidirectional 1:1 relations to other domain objects
- unidirectional 1:n relations to other domain objects
- bidirectional 1:1 relations to other domain objects
- bidirectional 1:n relations to other domain objects

The differences and finesse of modeling a domain with these relations is explained in PhoneBook.Domain -- relationships