Database Handling Through Custom Types

Göran Törnquist

Introduction

Lassoscript allows you to access databases through inlines. The specifics of those databases are setup once through the administrative interface, and then we refer to them using the inlines. This way of deferring settings and then connecting functionality to them is called abstraction of the database handling.

This paper is about how to use Custom Types to make handling of records in database tables. The start is a basic custom type which will show how to create a custom type as such. In the end of the paper, you have a small set of custom types that solves a number of issues that the web developer has deal with on a daily basis.

It is the intention that you will see the benefits from using custom types together with database access to make your database implementation easy to maintain and easier to test for quality assurance. While covering the custom types part, you will also learn some about custom tags.

This covers one way of approaching database access in a object oriented way. It is not meant to be a complete guide to object orientated programming, neither does it employ the perfectly secure implementation of database access - we will actually bypass the Lasso database security scheme as much as possible to make things simpler.

Securing the code will be left as an exercise to the reader since it would be a too complex subject to cover in such a short time. In other words, I've kept the implementation as simple as possible to focus on the custom type and database part.

Error handling has also been left out to leave space for conceptually important code.

If you're more experienced, skip to the last two sections. Otherwise, please read on while I walk you through constructing a simple custom type.

What is a custom type?

Custom types, or commonly ctypes, are the way Lassoscript allows you to define your own data types instead of being limited to the types that are defined in Lassoscript from the start.

Let's look at a custom type using an example:

A rectangle can be defined to cover four points in a coordinate system; a left, top, right and bottom position.

```
<?Lassoscript
Define_Type: 'Rectangle';
  local: 'top' = 0;
  local: 'left' = 0;
  local: 'bottom' = 0;
  local: 'right' = 0;
```

```
/Define_Type;
var: 'myRectangle' = Rectangle;
//creates an instance of the custom type
?>
```

It's great to be able to create a variable that automatically has the four points needed to define the rectangle. But a real world example will prove to us that we need to find out the width, length and area of the rectangle.

This is the procedural way of solving that need:

```
<?Lassoscript
Define_Tag: 'rectangleWidth', -required='left', -required='right';
  return: (math_abs: #right - #left);
/Define_Tag;
$myRectangle->'right' = 100;
$myRectangle->'bottom' = 50;
var: 'myRectangleWidth = (rectangleWidth: $myRectangle->'left', $myRectangle-
>'right');
?>
```

To me the object oriented way feels both easier and cleaner:

```
<?Lassoscript
Define_Type: 'Rectangle';
  local: 'top' = 0;
  local: 'left' = 0;
  local: 'bottom' = 0;
  local: 'right' = 0;
  Define_Tag: 'getWidth';
  return: (math_abs: self->'right' - self->'left');
  /Define_Tag;
/Define_Type;
var: 'myRectangle' = Rectangle;  //creates an instance of the custom type
$myRectangle->'right' = 100;
$myRectangle->'bottom' = 50;
var: 'myRectangleWidth = $myRectangle->getWidth;
?>
```

By defining the custom tag getWidth within the custom type we have been given a way to actively use the data defined in the custom type. This is commonly referred to as encapsulation. The retangle custom type "knows" how to operate on the data.

In due time the programmer will find out that someone might be feeding the custom type with erroneous data and then the rectangle ctype will evolve. Anywhere where we use the width calculation on the rectangle ctype will be benefiting from this change.

```
Define_Tag: 'getWidth';
  return: (math_abs: (integer: self->'right') - (integer: self->'left'));
/Define_Tag;
```

Will I produce faster and more compact code?

No. Abstraction and generic handling all comes with a cost. Since a ctype is concerned about a generic situation, everything that could occur has to be handled in a generic way. The upside is that it will possibly handle quite a few different situations. The downsides are that you will type more and the code will not be as easily optimized as in the procedural way.

So why use custom types at all?

The simple and quite generic answer is: To reach the goals for your web application in a way that makes it easy to produce, maintain and document your solutions.

Predictability: By using custom types, you will be able to create data types that always has a specific set of attributes and behave in a specific way. Therefore, the use of these will lead to results that are easier to predict.

Reusability: Thinking in an object oriented way often leads to generic solutions of problems. Used the right way, this can save lots of development time when similar problems needs to be solved in different parts of your application or applications.

Abstraction: When using an object oriented approach, you speak for example of People and how they behave themselves and relate to other things in your system.

Less complexity: Through abstraction, you can hide complicated tasks and constraints. A good example of this is the PDF_Doc custom type that let you produce a PDF file in relatively easy way. There are quite a few more examples that comes with Lasso from the beginning.

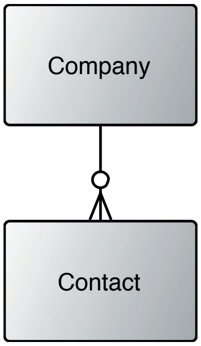
Why, in particular, use custom types for database access?

First of all, code controlling database access is very often created when needed. The need at the moment often defines how far you're thinking when accessing the database. If anything changes with the database definition or the need behind the database access, then you will need to go through every place where you've been accessing the database and walk through the process of creating or changing code, which leads to testing code, which leads to debugging code.

The database access is scattered throughout a web application, and we need a good way to make sure the data is there when we need it, and that it will be valid data. Custom types and database access paired together makes it possible to define the database access in one place, and then allow the scattered code to access the data in a well defined way.

The example

Custom types and the database modeling are forming a good relationship. This will easiest be shown through an ERD (Entity Relation Diagram).



I will use a simple example throughout this paper. It is a database containing contacts belonging to a company. The MySQL schema is given below.

```
CREATE TABLE `contact` (
  `key` bigint(20) unsigned NOT NULL auto_increment,
  `firstname` varchar(64) NOT NULL default '',
  `lastname` varchar(64) NOT NULL default '',
  `company_key` bigint(20) NOT NULL default '0',
  PRIMARY KEY  (`key`)
);
CREATE TABLE `company` (
  `key` bigint(20) unsigned NOT NULL auto_increment,
  `name` varchar(64) NOT NULL default '',
  PRIMARY KEY  (`key`)
);
```

I hope those of you that uses FileMaker or any other type of database will be able to read and recreate the database from the given schema.

A simple custom type

Taken from example01/code.inc

```
Define_Type: 'C_Contact';
  local: 'firstname' = '';
  local: 'lastname' = '';
  Define_Tag: 'setName', -required='firstname', -required='lastname';
  self->'firstname' = #firstname;
  self->'lastname' = #lastname;
  /Define_Tag;
  Define_Tag: 'getName';
  return: self->'firstname' + ' ' + self->'lastname';
  /Define_Tag;
/Define_Type;
```

This example outlines the definition of a simple ctype named Contact. From this moment on we will be able to define a variable to be a Contact.

```
var: 'speaker= Contact;
```

We can also start using it with the same syntax we use with the built in types.

```
$speaker->(setName: -firstname='Göran', -lastname='Törnquist');
```

compared to manipulation of a simple string variable

```
var: 'myName' = 'Göran Törnquist';
$myName->(replace: 'ö', 'o');
```

From the syntax viewpoint there is no difference between the code accessing the Contact and the String variables. Though, obviously, they are not equal in function or in contents. The conclusion is that there's nothing magic or special with ctypes. You can access them in the same way that you normally access any type of variable.

Accessing the attributes

Taken from example02/code.inc

```
Define_Type: 'C_Contact';
  local: 'key' = 0;//unique key to identify record
  local: 'firstname' = '';
  local: 'lastname' = '';

  //*** previous code for member tags taken out for reasons of space ***/
  Define_Tag: 'getKey';  //getting the the key instance variable
  return: self->key;
  /Define_Tag;

  /* You will rarely use the setKey method from outside of this custom type */
  Define_Tag: 'setKey', -required='key';  //setting the key instance variable
  self->'key' = (integer: #key);
  /Define_Tag;
/Define_Type;
```

The member variable 'key' is directly referring to the field that uniquely is used to store values to identify a specific contact. We use member tags to access the key because it makes our code less specific to the type of the key. Even though we won't likely change the type of the key, it is good to use a generic way of handling such a common feature of a database record. If we later create the company ctype, then it will be good to access the key of that ctype in a similar manner.

Retrieving data

To load the contact data from the database, we define a new member tag called 'load'. Since the object representing the record doesn't contain any valid data yet, we have to provide the 'load' member tag with a valid key to retrieve the data from the database.

This is how the 'load' member tag looks like.

Taken from example03/code.inc

```
Define_Tag: 'load', -required='key';
  inline: -search, $gDBspec,
  -table='contact', -keyfield='key', -keyvalue=(integer: #key), -maxrecords=1;
  records;
  self->'key' = (integer: (field: 'key'));
  self->'firstname' = (field: 'firstname');
  self->'lastname' = (field: 'lastname');
  /records;
  /inline;
/Define_Tag;
```

It looks very much like any other inline tag. So, what's so special with this? At the moment it's not about the inline itself, but what the inline contains. All of the database record data is copied to the member variables of the ctype.

Using the ctype data

The way to use the data from the ctype variable can be very much alike any other data used within a html page.

taken from example04/contact.htm

```
<form name="contact_form" id="contact_form" action="" method="post">
  <fieldset>
  <legend>Contact info</legend>
  <label for="inp_firstname">Firstname
  <input type="text"
  name="inp_firstname" id="inp_firstname"
  value="[$speaker->'firstname']" />
  </label>
  <label for="inp_lastname">Lastname
  <input type="text"
  name="inp_lastname" id="inp_lastname"
  value="[$speaker->'lastname']" />
  </label>
  </fieldset>
</form>
```

Using getters and setters

There are different schools telling you their truth about object oriented programming. One way to look at member variables is that you should never access them directly. The main reason for this is that all access to data or functions through the object should be abstracted (or controlled).

While this is generally a very good idea, it adds chores to the programmers todo list: Writing getters and setters as well as testing and debugging the same. Also, the overhead to call a member tag and to access a member variable is completely different. I haven't personally carried out any performance tests to find the difference, but I expect the additional overhead to be significant compared with the simple access of a variable.

In different integrated development environments (IDE's) like the Java editor in Eclipse, you'll find wizards to generate the equivalent to member tags for getters and setters of the member variables. As of now, there is no such feature in Lasso Studio for Eclipse. If there were, then we would have a predictable generator of Lassoscript code for getters and setters. Predictability means consequential behaviour, and such can easily be tested and measured. Knowing what we have and what's going to happen builds stability.

In the code supporting this paper I have chosen to access the member variables directly. There are good sides to this and there are bad sides. As long as you know which they are, you're ready to accept the consequenses.

Storing data

The process of storing data in the database is quite straightforward, and does not yet reveal any special features of object orientation. We will get back to this member tag later to make the handling more generic than at the moment.

Taken from example05/ctypes.inc

```
Define_Tag: 'save';
  if: self->GetKey == 0;  //this is an add operation
  inline: -add, $gDBspec, -table='contact', -keyfield='key', -keyvalue=self->GetKey,
  'firstname'=self->'firstname',
  'lastname'=self->'lastname';
  self->(setKey: -key=keyfield_value);  //it is important to set the key of the object
  /inline;
  else;  //this is an update operation
  inline: -update, $gDBspec, -table='contact', -keyfield='key', -keyvalue=self->GetKey,
  'firstname'=self->'firstname',
  'lastname'=self->'lastname';
  /inline;
  /if;
/Define_Tag;
```

Finding generic ways to solve common operations

Earlier we used the 'load' member tag for loading the contact data from the database. We are now going to rewrite it to allow for more generic implementation. This way we can reuse the code in other ctypes as well as test it thoroughly through use in other ctypes.

The new version of C_Contact->Load looks like this:

Taken from example06/ctypes.inc

```
Define_Tag: 'load', -required='key';
  inline: -search, $gDBspec,
  -table='contact', -keyfield='key', -keyvalue=(integer: #key), -maxrecords=1;
  records;
  self->LoadX;
  /records;
  /inline;
/Define_Tag;
```

Instead of specifying the fields directly in the C_Contact->Load we split the functionality between one generic member tag and one specific, called C_Contact->loadX. This way we know that most of the code that is specific to the contact table will be found in C_Contact->loadX.

```
Define_Tag: 'loadX';
  self->(setKey: -key=(field: 'key'));
  self->'firstname' = (field: 'firstname');
  self->'lastname' = (field: 'lastname');
/Define_Tag;
```

If you worry about the references to the contact table in C_Contact->load, please don't.We'll come back for another round of making the code generic.

Loading more than one contact

So far we've been concerned with only one contact, but we also need a way to present a list of contacts. It would be good if that code could be both stored with the rest of the code acessing the contact table. Luckily, the object oriented model provides us with a way to do that also.

Returning an array of contacts

The code to return an array of contacts is quite straight forward.

```
Define_Tag: 'getContactList';
  local: 'result' = array;

  local: 'oneRecord' = map;
  inline: -search, $gDBspec, -table='contact', -keyfield='key', -maxrecords='all';
  records;
  #oneRecord = C_Contact;
  #oneRecord->loadX;
  #result->(insert: #oneRecord);
  /records;
  /inline;
  return: #result;
/Define_Tag;
```

Using an array of contacts

Here follows the contents of the the file example06/list.htm which is included as an html fragment into the main html file.

```
<table id="contact_list">
  <tr>
  <th></th>
  <th class="key">Key</th>
  <th class="firstname">Firstname</th>
  <th class="lastname">Lastname</th>
  </tr>
  [iterate: C_Contact->getContactList, (var: 'oneContact')]
  <tr>
  <td><a href="?key=[$oneContact->GetKey]">Edit</a></td>
  <td class="key">[$oneContact->'key']</td>
  <td class="firstname">[$oneContact->'firstname']</td>
  <td class="lastname">[$oneContact->'lastname']</td>
  </tr>
  [/iterate]
</table>
```

You can see that we're not even creating a variable to access the member tag. This way of using an custom type is very common in pure object oriented languages such as Java, though there the ctypes are called classes.

This way of accessing the code works because Lasso will internally create a temporary variable from the ctype definition and after it's been used - i.e. no code is referring to it - the temporary variable will be forgotten.

Increasing the efficiency of instantiation of custom types

Whenever a custom type is used to create a variable, it will run all the code between the [Define_Tag] and the [/Define_Tag]. The reason for this is that earlier versions of Lassoscript was depending on code being executed at the time of the creation.

Now, much time has passed and the object oriented support in Lassoscript has grown to a more mature version. Today the only code that is expected to be found in [Define_Tag]...[/Define_Tag] is definitions of locals, which are called member variables, and definitions of member tags.

Since we're supporting this more modern way of declaring the ctype, we let Lasso find that out by using the special keyword -prototype.

It looks like this:

```
Define_Type: 'C_Contact', -prototype;
```

This will tell Lasso to immediately create an invisible variable which will be copied whenever we create a new instance of the C_Contact ctype. By doing that the speed will increase more than tenfold when used with ctypes with a large amount of code. You will find out that ctypes tends to be defined by a lot of code - generic code demands that you take a lot of possible cases in account.

Delete a record

To delete a contact we need a C_Contact->delete member tag. It looks almost as the C_Contact->save, except it doesn't make any use of parameters.

Taken from example08/ctypes.inc

```
Define_Tag: 'delete';
  if: self->hasValidKey;  //this is an add operation
  inline: -delete, $gDBspec, -table='contact', -keyfield='key', -keyvalue=self->GetKey;
  self->(setKey: -key=0);  //it is important to reset the key of the object
  //so that it is considered invalid
  /inline;
  /if;
/Define_Tag;
```

The generic question

In C_Contact->delete we use a member tag for questioning the validity of the record key. By doing this we will know that all code, no matter where it's being used will define the invalid key in the same way.

Taken from example08/ctypes.inc

```
Define_Tag: 'hasValidKey';
  return: self->key > 0;
/Define_Tag;
```

This kind of questioning of status or state of member variables is encouraged since the code uses it becomes very easy to read. See also the paragraph about getters and setters.

Compare

```
if: self->key > 0;
```

with

```
if: self->hasValidKey;
```

The latter version is easier to read and has the advantage that no matter the type of key that the record uses, the code will still look the same from the callers' perspective.

What we have done here is to introduce a questioning action in the ctype. With the references to human languages this is often called a predicate.

Hiearchy and inheritance in general

When a ctype is based on another ctype, we use a hierarchical way of defining our conceptual model of the world. Before we start adapting the C_Contact ctype to be using inheritance to pass over the generic code to a base ctype, we'll take a look at a totally different case.

Base custom type

Let's consider a geometric application which needs two different types of shapes: the rectangle and the ellipse.

We could define them like this

```
Define_Constant: 'pi', 3.14;
Define_Type: 'Rectangle';
  local: 'top' = 0.0;
  local: 'left' = 0.0;
  local: 'bottom' = 0.0;
  local: 'right' = 0.0;
  Define_Tag: 'getWidth';
  return: (math_abs: (decimal: self->'right') - (decimal: self->'left'));
  /Define_Tag;
  Define_Tag: 'getHeight';
  return: (math_abs: (decimal: self->'bottom') - (decimal: self->'top'));
  /Define_Tag;
  Define_Tag: 'getArea;
  return: self->getWidth * self->getHeight;
  /Define_Tag;
/Define_Type;
Define_Type: 'Ellipse';
  local: 'top' = 0.0;
  local: 'left' = 0.0;
  local: 'bottom' = 0.0;
  local: 'right' = 0.0;
  Define_Tag: 'getWidth';
  return: (math_abs: (decimal: self->'right') - (decimal: self->'left'));
  /Define_Tag;
  Define_Tag: 'getHeight';
  return: (math_abs: (decimal: self->'bottom') - (decimal: self->'top'));
  /Define_Tag;
  Define_Tag: 'getArea;
  return: self->getWidth * self->getHeight * pi;
  /Define_Tag;
/Define_Type;
```

But I'd rather define them like this

```
Define_Constant: 'pi', 3.14;
Define_Type: 'Shape';
  //the position of the boundary box is defined below
  local: 'top' = 0.0;
  local: 'left' = 0.0;
  local: 'bottom' = 0.0;
  local: 'right' = 0.0;
  Define_Tag: 'getWidth';
  /Define_Tag;
  Define_Tag: 'getHeight';
  /Define_Tag;
  Define_Tag: 'getArea;
  /Define_Tag;
/Define_Type;

Define_Type: 'Rectangle', 'Shape';
  Define_Tag: 'getWidth';
  return: (math_abs: (decimal: self->'right') - (decimal: self->'left'));
  /Define_Tag;
  Define_Tag: 'getHeight';
  return: (math_abs: (decimal: self->'bottom') - (decimal: self->'top'));
  /Define_Tag;
  Define_Tag: 'getArea;
  return: self->getWidth * self->getHeight;
  /Define_Tag;
/Define_Type;

Define_Type: 'Ellipse', 'Shape';
  Define_Tag: 'getWidth';
  return: (math_abs: (decimal: self->'right') - (decimal: self->'left'));
  /Define_Tag;
  Define_Tag: 'getHeight';
  return: (math_abs: (decimal: self->'bottom') - (decimal: self->'top'));
  /Define_Tag;
  Define_Tag: 'getArea;
  return: self->getWidth * self->getHeight * pi;
  /Define_Tag;
/Define_Type;
```

Please note the empty declarations of getWidth, getHeight and getArea in the Shape ctype. These are there as placeholders so that code that refers to any type of Shape also will be able to perform without error.

When you declare a variable of the type Rectangle, you have to understand that a rectangle is not only a rectangle, but also a Shape.

```
var: 'myRectangle' = Rectangle;
$myRectangle->'right' = 100.0;
$myRectangle->'bottom' = 50.0;
var: 'myArea' = $myRectangle->getArea;
```

The example here will return 5000.0 as the area since it appears frontmost to be a rectangle. But all the calculations are done on the member variables of the Shape. So it seems you have access to everything of the ancestor ctype called Shape.

The space here is too limited to go into the real depths of inheritance, shadowing, overloading and quite a few more object oriented paradigms. If this seems interesting to you, there are a lot of books on the subject.

The base custom type

We have reached a point where we can start to discuss our base custom type. The foundation for the database access through custom types. To get there with the whole picture, we'll skip to example12 in the supporting material.

The goal

First of all, where are we going with the C_Contact custom type? We're limiting it to only consist of attributes and actions that has to be defined within the concept of a contact. The rest is either part of the base ctype or is a special form of Contact itself - a successor to the C_Contact, e.g. C_Salesperson or something along that line.

Taken from example12/ctypes.inc

```
Define_Type: 'C_Contact', 'C_Record', -prototype;
  local: 'table' = 'contact';
  local: 'keyField' = 'key';
  local: 'firstname' = '';
  local: 'lastname' = '';

  Define_Tag: 'loadX';
  self->'firstname' = (field: 'firstname');
  self->'lastname' = (field: 'lastname');
  /Define_Tag;

  Define_Tag: 'save';
  self->parent->(save:
```

```
  'firstname'=self->'firstname',
  'lastname'=self->'lastname'
  );
  /Define_Tag;

  Define_Tag: 'setName', -required='firstname', -required='lastname';
  self->'firstname' = #firstname;
  self->'lastname' = #lastname;
  /Define_Tag;

  Define_Tag: 'getName';
  return: self->'firstname' + ' ' + self->'lastname';
  /Define_Tag;
/Define_Type;
```

The amazing part is that this is the full implementation of the C_Contact in our example. It would be very easy to define a company ctype to complement the C_Contact.

Another ctype created in virtually no time

Let's show the C_Company ctype would look like:

Taken from example12/ctypes.inc

```
Define_Type: 'C_Company', 'C_Record', -prototype;
  local: 'table' = 'company';
  local: 'keyField' = 'key';
  local: 'name' = '';

  Define_Tag: 'loadX';
  self->'name' = (field: 'name');
  /Define_Tag;
  Define_Tag: 'save';
  self->parent->(save:
  'name'=self->'name'
  );
  /Define_Tag;

/Define_Type;
```

What we have done here is to define the ctype, add the member variables to show what is contained in the database table, and specified how we would like these values to be loaded and saved from the database.

The ancestor of all database records

To have our C_Contact and C_Company ctypes working we need the base ctype defined. This is what you would call an abstract class in OOP lingo. It means that there will never be a variable directly defined being this type. There will instead be C_Contact based on C_Record and C_Company based on C_Record.

Dissection of the C_Record custom type

This section is a bit differently structured than the previous ones. It is merely a wrap up of what has been covered in the past pages. Some parts are quite advanced, some are less.

```
Define_Type: 'C_Record', -prototype;
  local: 'key' = null;
  local: 'table' = '';
  local: 'keyField' = '';
  local: 'keyType' = 'integer';
```

The member variables here are telling us what a record consists of. Since there is no concept of a database or a table in this example, I have chosen to include the table name in the record, but left out the database and authentication information. The latter decision has been done to leave place for a Roundtable discussion.

```
  Define_Tag: 'castKey', -required='key';
  return: ((pair: (\(self->'keyType'))=(array: #key))->invoke);
  /Define_Tag;
```

The C_Record->castKey member tag is what makes the record being independent of the type of the keyfield. It will on the fly cast the key that has been passed as a parameter to the specified keyfield type. If we would like to get really in to the depths, this could also be a ctype itself, which would handle more advanced type such as non-predictable record keys.

```
  Define_Tag: 'getKey';  //getting the the key instance variable
  return: self->key;
  /Define_Tag;

  /* You will rarely use the setKey method outside the custom type hierarchy */
  Define_Tag: 'setKey', -required='key';  //setting the key instance variable
  self->'key' = (self->(isValidKey: #key) ? self->(castKey: #key) | null);
  /Define_Tag;
  Define_Tag: 'hasValidKey';
  return:  self->(isValidKey: self->'key');
  /Define_Tag;

  Define_Tag: 'isValidKey', -required='key';
  return: #key != null && ((pair: (\(self->'keyType'))=(array: #key))->invoke) != 0;
  /Define_Tag;
```

The C_Record->getKey, C_Record->setKey, C_Record->hasValidKey, and C_Record->isValidKey forms the interface to access the key and to validate the key in a generic manner. This is a good example of when getters and setters are great to use.

```
  Define_Tag: 'load', -required='key';
  if: self->(isValidKey: #key);
  inline: -search, $gDBspec->getSpec,
  -table=self->'table', -keyfield=self->'keyfield', -maxrecords=1, -keyvalue=#key;
  records;
  self->(setKey: -key=(field: self->'keyfield'));
  self->LoadX;
  /records;
  /inline;
  /if;
  /Define_Tag;
```

The C_Record->load tag does the actual job of accessing the database to locate the record. It is the equivalent of the [inline: -search] container. Since the key handling is a instrumental part of the generic handling of a record, we have put the responsibility to set the key value on the C_Record ctype whenever it can be done.

```
  Define_Tag: 'loadX';
  /Define_Tag;
```

The C_Record->loadX tag is the supporting tag that actually performs the copying of data from the database to the abstracted database record.

```
  Define_Tag: 'save';
  if: self->hasValidKey;  //this is an update operation
  inline: -update, $gDBspec->getSpec,
  -table=self->'table', -keyfield=self->'keyfield', -keyvalue=self->GetKey, params;
  /inline;
  else;  //this is an add operation
  inline: -add, $gDBspec->getSpec,
  -table=self->'table', -keyfield=self->'keyfield', -keyvalue=self->GetKey, params;
  self->(setKey: -key=keyfield_value);  //it is important to set the key of the
object,
  //so the live object is considered valid
  /inline;
  /if;
  /Define_Tag;
```

The C_Record->save tag has a little bit of a different implementation than the C_Record->load tag. The reasons for this is that we might be wanting to save different fields of the database in different situations. However in this implementation, a loaded successor of C_Record must be fully loaded to be considered valid.

```
  Define_Tag: 'delete';
  if: self->hasValidKey;  //this is an add operation
  inline: -delete, $gDBspec->getSpec,
  -table=self->'table', -keyfield=self->'keyfield', -keyvalue=self->GetKey;
  self->(setKey: -key=null); //it is important to set the key of the object
  //so that it is considered invalid
  /inline;
  /if;
  /Define_Tag;
```

The C_Record->delete tag is the opposite of the C_Record->save tag. The most important part is to invalidate the key so that we know that the live object is not associated with a database record anymore. It is therefore perfectly possible to do a [$spokesman->delete] followed by a [$spokesman->save]. Most likely that would have no use in the real world, but the essence is that we can trust a deleted record to be fully represented as deleted.

```
  Define_Tag: 'getList';
  local: 'result' = array;

  local: 'oneRecord' = map;
  inline: -search, $gDBspec->getSpec,
  -table=self->'table', -keyfield=self->'keyfield', -maxrecords='all';
  records;
  #oneRecord = (\(self->type))->Invoke;
  #oneRecord->(setKey: -key=(field: self->'keyfield'));
  #oneRecord->loadX;
  #result->(insert: #oneRecord);
  /records;
  /inline;
  return: #result;
  /Define_Tag;
```

The C_Record->getList tag is the foundation for a general type of loading more than one record from a database. For it to be useful you'd need to define different sorting orders. For performance sake you'd need to define partial loading etc. The technique used to instantiate the ctype is one of the major strengths behind dynamically typed object oriented languages.

```
/Define_Type;
```

Summary

Once you start using ctypes in this way you'll be stuck into the lazy life of expecting things to behave themselves. You are allowed not only to define custom tags that are used to extend the functionality of Lasso, but also to build new conceptual models of "things" and "objects". Most likely you'll start to speak about the database records in a different manner.

This is the start of a two-tier database solution. Your next step would be a three-tier solution, which can be constructed upon a well built two-tier solution.