Version 5.5

Web Application Example

MetaCase Document No. WA-5.5

Copyright © 2018 by MetaCase Oy. All rights reserved

First Printing, 2nd Edition, August 2018

MetaCase
Ylistönmäentie 31
FI–40500 Jyväskylä
Finland

Tel: +358 400 648 606
E-mail: info@metacase.com
WWW: http://www.metacase.com

No part of this manual may be reproduced or transmitted in any form or by any means, electronic or mechanical, including but not limited to photocopying, without express written permission from MetaCase.

MetaEdit+ is a registered trademark of MetaCase. The other trademarked and registered trademarked terms, product and corporate names appearing in this manual are the property of their respective owners.

Preface

The web application example illustrates how UML can be used (and abused!) to specify web applications, and how complete working applications can be produced from those models. To achieve this, a domain-specific generator is implemented into MetaEdit+. Unlike the standard generators supplied with most UML tools, which can only create class and method skeletons, this generator produces the full code that we want for each application.

Using the example, a developer can design simple database web applications using the familiar core concepts of UML. We will also see how going beyond simple examples requires increasing amounts of twisting and redefining of UML semantics, resulting in a language that is hard to use, and no longer "UML" in anything but notation.

Normally, a web application would be written in a server-side programming language. Since you might not have a server handy, or perhaps don't have the right to upload applications and create databases, we will make this web application run purely in your browser. We will create HTML5 pages for the user interface, JavaScript for the behavior, and use an SQL database running in the browser for the back end.

To explore the web application example thoroughly, the following things are required:

- MetaEdit+ for trying out the web application language. For further information about MetaEdit+, please refer to the MetaEdit+ User's Guide.
- The web application patch, to add web application generators and example models to the 'UML' project in the MetaEdit+ demo repository.
- A web browser supporting HTML5 Web SQL Database (currently Chrome, Safari or Opera), with JavaScript support enabled.

We expect that you have a working knowledge of MetaEdit+. If you want to extend the modeling language or generators further you should have MetaEdit+ Workbench or the evaluation version available from http://www.metacase.com.

1 Web application example

The web application example breaks the rules of DSM – but in a good cause. Most developers have seen UML used for specifying databases: classes map to tables and attributes to columns. What UML can't do so well is specify behavior: even if we enter the names and parameters of operations, or add a state diagram, we have only shown how we want to break that behavior down into chunks, but still UML has not helped us create the contents of those chunks. But what if we could do without behavior altogether? Or to be more precise, if we could specify behavior once for a whole class of applications, and have each application follow that behavior guided by its own data? If that were so, even the humble UML class diagram might be enough to specify any of a range of useful applications.

Can an application's behavior come from data? Of course, we are all familiar with this on a simpler level. In some way, all applications are guided by their data: if we want to show the surname of a person, we need to display different characters depending on whether the person is John Smith or Jane Doe. The code doesn't contain the data, just the variable names which are replaced by data at runtime:

```
writeln("First Name: " + firstname);
writeln("Surname: " + surname);
writeln("Age: " + age);
```

We can go a step further, though: rather than writing code like the above, we can write generic code that says "print all attributes and their values":

```
for (i=0; i<attributes.length; i++){
  writeln(attributes[i] + ": " + values[i];
}
```

With code like that, we can supply any number of any kind of attributes, and things will just work. For example:

```
var attributes = [
  {name:"First Name", datatype:"string"},
  {name:"Surname", datatype:"string"},
  {name:"Age", datatype:"int"}
];
var values = ["John", "Smith", 43];
```

We thus have data (`values`) but also data about that data (`attributes`). Data about data is often called meta-data, and programming that takes advantage of it is called meta-data programming. With a little meta-data programming we can specify the generic behavior of simple database web applications once, and let the rest be generated from the models.

In this first chapter we show how to install the web applications example and try it out on a sample model to see how the generated application works in practice. Chapter 2 looks at how UML is used for this example, and the limitations of UML when going beyond simple web applications. Chapter 3 describes how the generator and domain framework were defined.

Please note that testing the modeling language and models presented here requires basic knowledge on how to use MetaEdit+.

1.1 OPENING THE WEB APPLICATIONS EXAMPLE

Installing the Web applications example will add some generators to UML Class Diagrams and two graphs, Football and Ordering, to the UML project.

Open MetaEdit+, select the demo repository, select the UML project, and press Login. If you are using the evaluation version and this is your first login, you will be asked to enter the evaluation code you received in your evaluation email.

1.2 AN EXAMPLE

Select UML in the Projects list and double-click Football in the Graphs list to open the following Class Diagram:

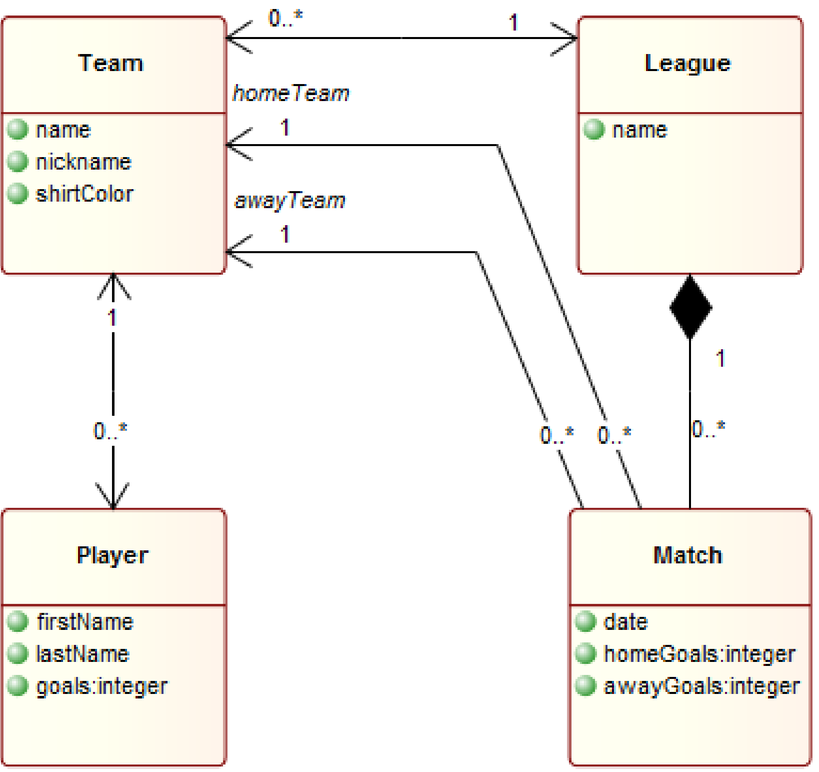


Figure 1-1. Football league web application

Choose **Graph | Generate…** and under “Class Diagram [UML] generators” choose “Web Application”. The files will be generated and your browser will open on the home page of the application.

If the browser that opens does not support Web SQL Database (e.g. currently Firefox and Internet Explorer do not), you will see a red warning in the page stating this. Install Chrome, Safari or Opera, set it to be your default browser, and run the generation again.

On the home page of the application you will see the same UML model as above, and can click any of the classes there to go to a page for entering and editing those elements. For now, click the League class, and you will see a page where you can enter a name for your league.

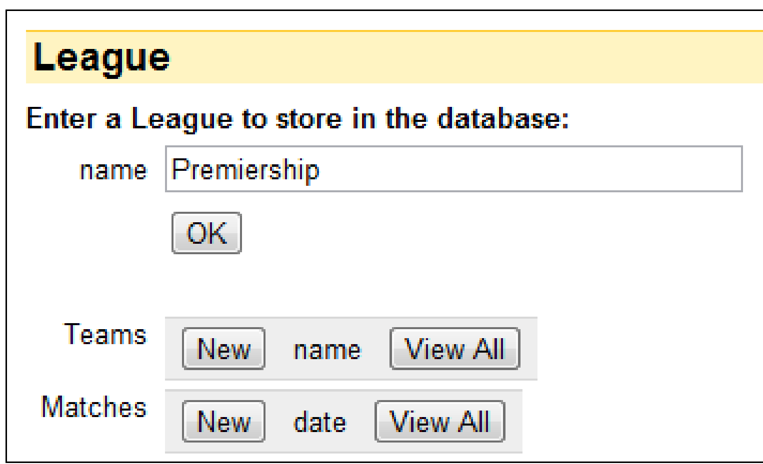


Figure 1-2. Creating a League

Type "Premiership" and press OK. The form will remain, allowing you to create more leagues, but one is enough for now. Below the OK button you will see that a league has a list of teams, but that is currently empty. Similarly, there is a list of Matches, also empty. The fact that a league has a name comes from the "name" attribute in the "League" class in our UML diagram. The lists of Teams is included because a League has an association to zero to many Teams. Similarly, a League is a composite aggregation of many Matches.

Press the New button next to Teams to go to the page for creating and editing teams. There you will see the reverse of the association between League and Team: each Team is in exactly one League. The field for League is thus shown in your browser as a pull-down menu, from which you can choose any existing league. You can then fill in the name, nickname and shirt color for this team, and press OK. You might want to create a couple more teams while you're here.

In the same way as each League had many Teams, each Team has many Players, and by pressing the New button next to Players you can start creating some players. The Team for each new Player is chosen from the pull-down menu on the Player page. (This may seem counter-intuitive if you are used to object-oriented thinking, where a Team would have a list of Players. In relational databases there is no "list" data type, so a Team does not refer to Players at all; instead, each Player refers to its Team by a unique name or ID of the Team.)

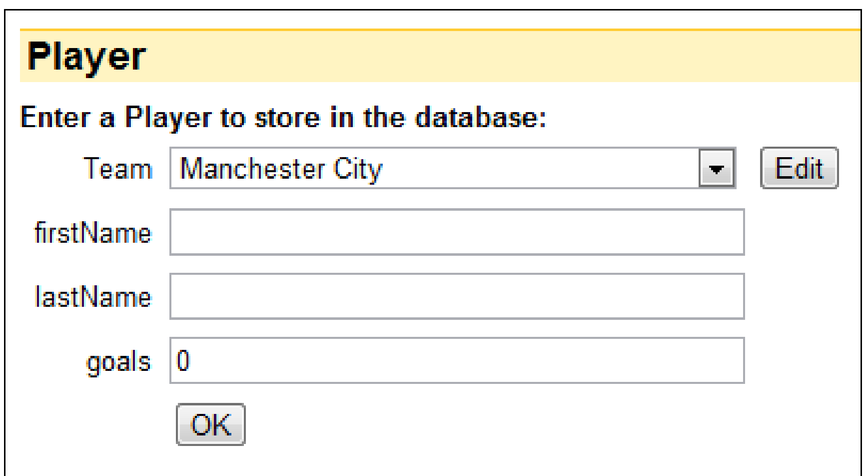


Figure 1-3 Creating a Player

You will notice that the "goals" field contains a zero. That is because Player's "goals" Attribute is defined with data type "integer" in the model. If no data type is specified in the

model, a simple string is assumed. You can also specify a default value in the Attribute; there is no need to quote strings there, as all values are simply placed in the field as-is.

After creating some players for the teams, you might have something like this:

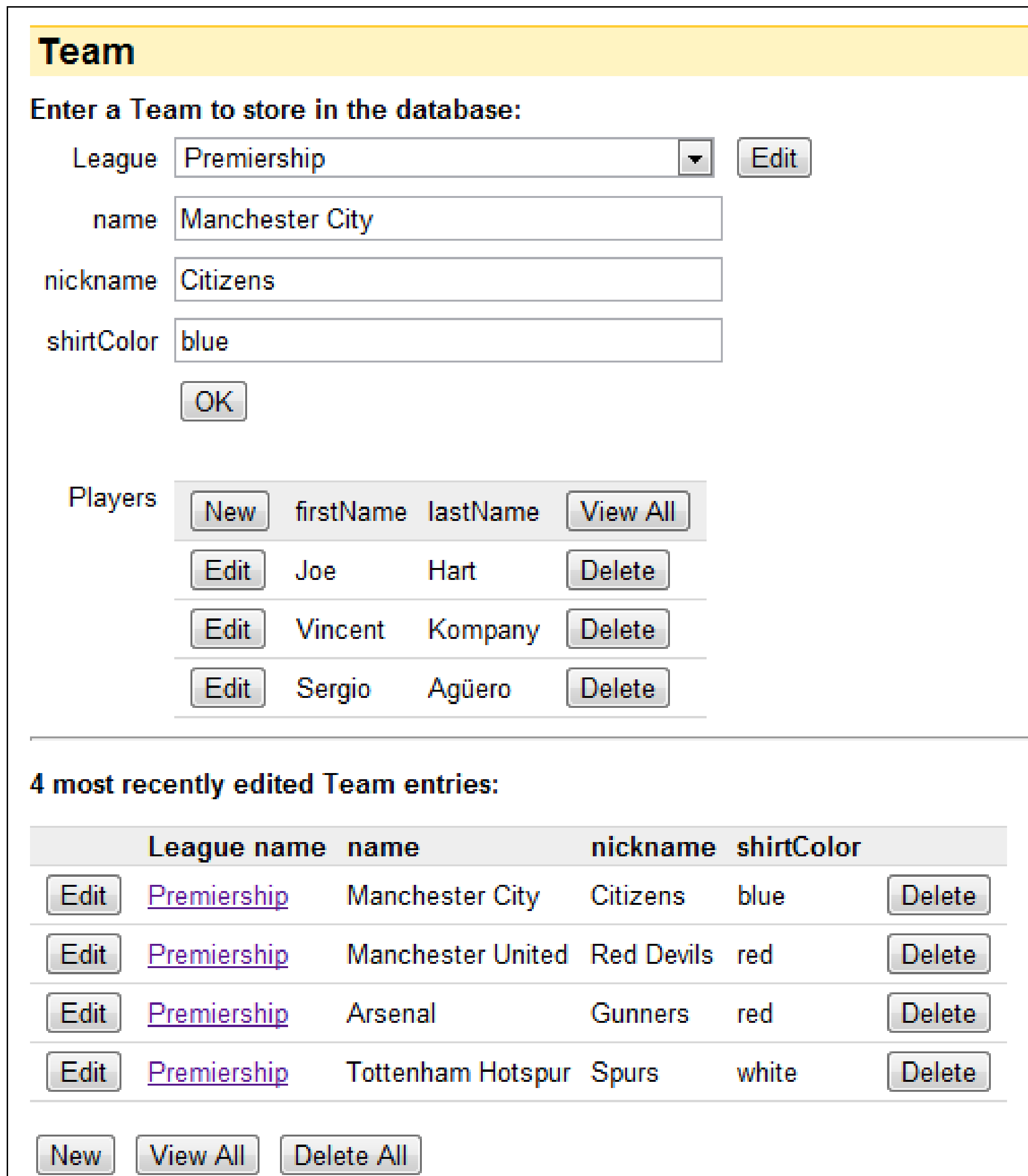


Figure 1-4. A Team and its Players

As you can see, the team now lists its players, with buttons to jump to edit a player. At the bottom of the screen is a handy list of the most recently edited teams, each with an Edit button so you can quickly jump to it. Since one of the pieces of information about a Team, its League, is a reference to another page, that shows up in the list as a link. Below the list are buttons to create a new team, view the full list of teams, or delete all teams.

Because all the data you enter is stored by your browser in a local SQL database, if you close your browser the data is still there. All your teams and players will appear again when you open those pages, e.g. by generating again or simply going to the HTML page. (Each browser keeps its own database, so if you switch browsers your data will not appear in the other browser.)

2 How UML is used for these web applications

The core modeling concepts of UML are hopefully familiar. We only use Class Diagrams, and within them only Classes, their Attributes, Associations and Aggregations; inheritance is ignored. Attributes can have data types of string (the default, mapping to varchar(255)) or integer (int is also accepted), and can specify default values (with no quotes). Relationships are always one-to-one or one-to-many, i.e. at most one end can specify a cardinality greater than one (0..*, 1..*, *). An Association role or Part must be marked as Navigable for that link to show up in the web forms; Association roles show an open arrow when Navigable, Parts do not. (At least not in standard UML notation: with MetaEdit+ you can of course change the Part symbol definition.)

2.1 FINE TUNING THE USER INTERFACE BY ABUSING UML

Defining association or aggregation relationships adds extra elements to the UI to show the items linked by that relationship. For examples, see the League field at the top of Figure 1-4 or the Players table there. By default, a linked item will show up in the tables as its first Attribute, e.g. the “name” of a Team or “firstName” of a Player.

To show more details, we need to find somewhere in the models to specify what to show for each relationship. Conveniently, the roles of those relationships have a property, Qualifiers, that allows you to specify Attributes that qualify the relationship. According to the UML specification, a qualifier is intended to provide an index or key for the set of related elements. Although that is not the semantics we want, this is the only place where we could record the information we need, so we will abuse UML for our own needs. This is a hack to achieve what we want, at the expense of making the models no longer standard UML, and requiring users to learn the new and unfamiliar semantics we attach to the familiar UML concepts.

You can thus specify which attributes you want to show in the fields and tables of the UI for the linked element by adding Qualifiers on the role to that element. For example, the 0..* Association role to Player from Team specifies “firstName” and “lastName” as qualifiers, so in Team’s table of Players in Figure 1-4, each Player’s first and last names are shown. To add new Qualifiers to a role, double-click the role line to open its Properties dialog, open the pop-up menu in the Qualifiers box and choose **Add Element…**. For our purposes, it is enough that you type the same name for the Qualifier attribute as for the Attribute in the Class.

If you want to do things perfectly, you could instead choose **Add Existing…**, which will let you refer to the exact Attribute in the Class – protecting you against any later changes to its name. **Add Existing…** will show the following dialog, listing all existing attributes.

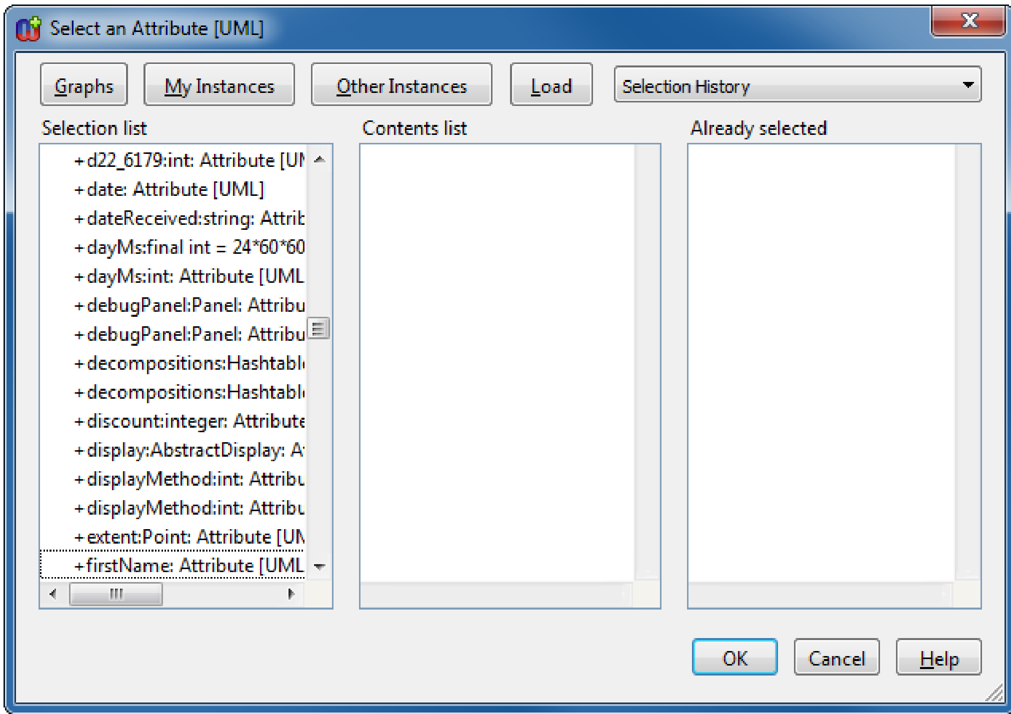


Figure 2-1. Adding an existing Attribute as a Qualifier

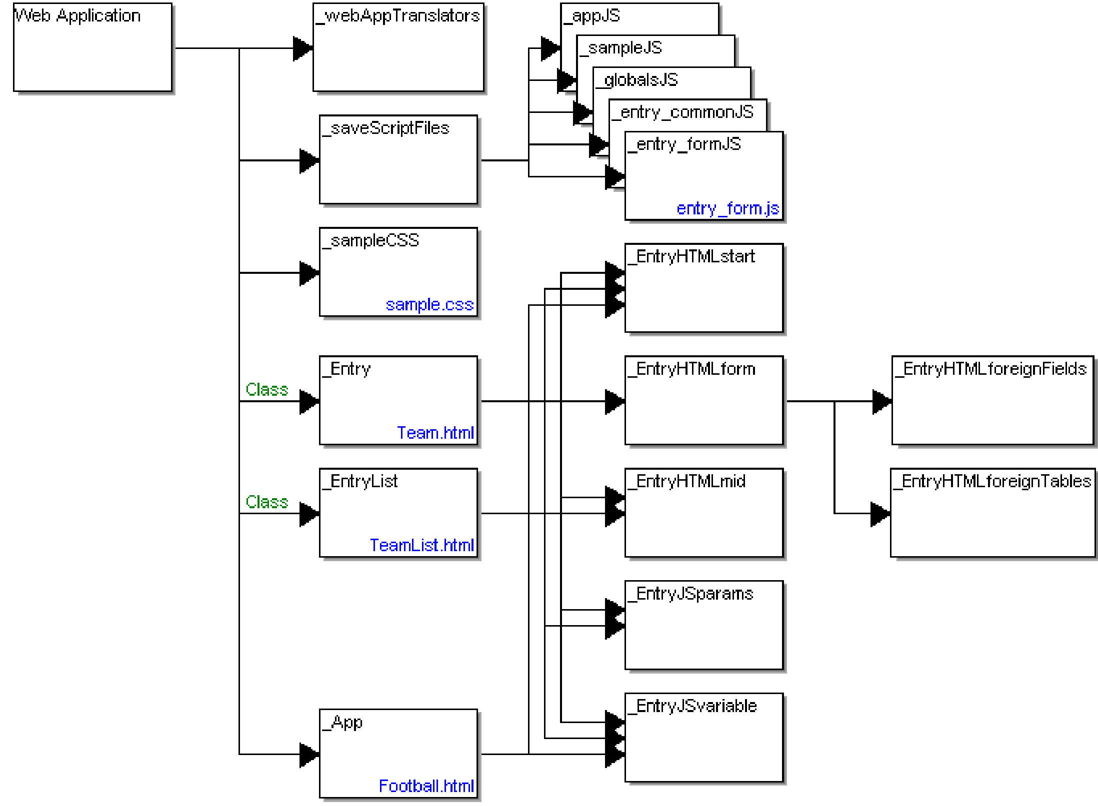
Since there are many models in the UML project, this list is quite long. The easiest way to get to an attribute is to type ahead in the left Selection list: if you type "+first" (the + is the prefix to show a public attribute), the selection in the list moves down to the first matching attribute. Double-clicking it will add it to the right-hand list, the collection of all the attributes you want to add in the Qualifier. You can find the other attributes by typing ahead again; if you mistype, just press Space to reset the type ahead buffer. Other ways of finding objects in this window include navigating via Graphs or Other Instances, and the Selection History list entries which work like bookmarks of the places you have recently added existing elements from. See the Component Selection Tool section in the MetaEdit+ User's Guide for more details.

3 Generators and domain framework

To best understand a modeling language, you need an example model. Similarly, to best understand a generator, you need the code generated from an example model. We will look here at the Football example, which is composed of the following files:

File	Description	Same for all apps?
Football.html	The home page for the application, with a model screenshot	No
Team.html	The main HTML page for Teams with the data entry form	No
TeamList.html	The HTML page to list all Teams	No
**.html, *List.html*	*Corresponding HTML pages for Player, League and Match*	*No*
sample.css	The Cascading Style Sheet	Yes
sample.js	Google's utility functions and Gears installation prompt	Yes
globals.js	JavaScript global variables and functions	Yes
entry_common.js	JavaScript functions for form and list pages	Yes
entry_form.js	JavaScript functions for form pages	Yes
app.js	JavaScript functions for the home page	Yes

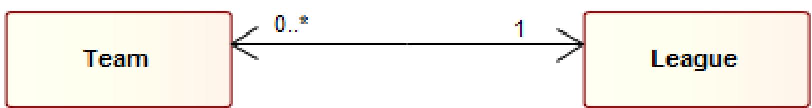
The generators and their subgenerators are shown below. Team.html is generated by `_Entry` and its subgenerators – and similarly for each class: Player, League and Match. TeamList.html is generated by `_EntryList`. Football.html is generated by the `_App` generator.



We will concentrate on Team.html, as seen in Figure 1-4. At the start of Team.html is the form for entering the data for a team. The list of 4 recent entries at the bottom of the page is largely reused as the content of TeamList.html. The rest of the HTML file includes the script files above, and provides the meta-data and code needed for Teams. Below, we will go through these sections in more detail, showing the link between the model, generators, and generated code.

3.1 REFERENCES TO OTHER PAGES

First come any fields that are references to other pages, e.g. the League field:



Each Team has a reference to one League, so the page for Team includes a list for selecting the appropriate League, along with a button to jump to the page to edit that League. The HTML for these is generated by `_EntryHTMLforeignFields`, resulting in the following:

League Premiership Edit

```
<td class="foreignLabel">League</td>
<td>
  <select id="fk_League" style="width:20em;"></select>
</td>
<td>
  <button type="button"
          onclick="editForeign('fk_League');"
          id="fk_LeagueButton">
     Edit
  </button>
</td>
```

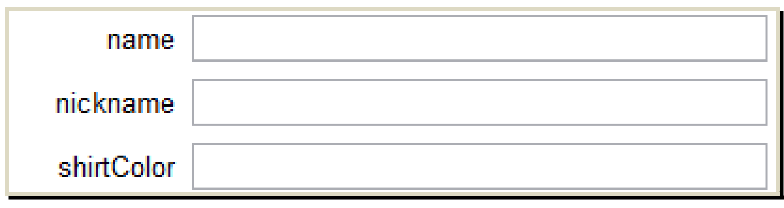
The set of choices in the field will be set later by JavaScript. The meta-data specifying the links to other pages is generated later by `_EntryJSVariable` into global variables `wholeForeign` and `assocForeign`, which are in turn concatenated into `allForeign`. In this case, there are no Aggregation / Whole-Part relationships, so `wholeForeign` is empty.

```
// Links from this table to another table by a foreign key
wholeForeign = [];
assocForeign = [
  {name:"fk_League",pageName:"League",foreignCols:["name"]}
];
```

3.2 LOCAL FIELDS ON THIS PAGE

Next come the fields for Attributes defined in this Class:

`_EntryHTMLform` generates a table row, label and input field for each Attribute:



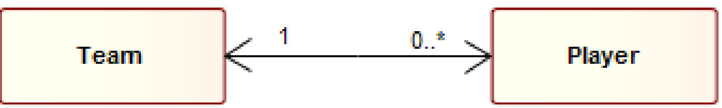

```
<tr>
  <td class="label">name</td>
  <td valign="middle">
    <input type="text" id="name" style="width:20em;">
  </td>
</tr>
<tr>
  <td class="label">nickname</td>
  <td valign="middle">
    <input type="text" id="nickname" style="width:20em;">
  </td>
</tr>
<tr>
  <td class="label">shirtColor</td>
  <td valign="middle">
    <input type="text" id="shirtColor" style="width:20em;">
  </td>
</tr>
```

The meta-data specifying the local columns in this database table is generated later by `_EntryJSVariable` into global variable `cols`.

```
// Simple local columns in this table
cols = [
  {name:"name",datatype:"varchar(255)",defaultValue:""},
  {name:"nickname",datatype:"varchar(255)",defaultValue:""},
  {name:"shirtColor",datatype:"varchar(255)",defaultValue:""}
];
```

3.3 REFERRERS TO THIS ENTRY FROM OTHER PAGES

Each Team has many Players:



Below the input fields, `_EntryHTMLforeignTable` generates the HTML for tables listing such one-to-many linked entries:

Players New firstName lastName View All

```
<tr>
  <td class="foreignTableLabel">Players</td>
  <td>
    <table class="foreignTable" id="PlayerTable">
      <tbody></tbody>
    </table>
  </td>
</tr>
```

In the generated HTML, there is only an empty placeholder table: the content, including the buttons and column headers, is filled in at runtime by the JavaScript function `displayForeignRows` in `entry_form.js`. Although the header row could have been generated as static HTML, the meta-data about the columns is needed anyway for filling in subsequent rows. It was thus easy enough to use the same code and meta-data to create the header row too.

The meta-data specifying the foreign key columns in other database tables that point to this table is generated later by `_EntryJSVariable` into global variable `referredAsForeign`.

```
// Links to this table from another table as a foreign key
referredAsForeign = [
  {name:"fk_Team", pageName:"Player", tableName:"tbl_Player",
   foreignCols:["firstName", "lastName"]
  }
];
```

3.4 LIST OF RECENTLY EDITED ENTRIES OF THIS TYPE

_EntryHTMLmid generates the list of 4 most recently edited Team entries as an empty placeholder table, along with its buttons to create a **New** Team or **View All** or **Delete All** Teams. Again, the contents of the placeholder tables will be filled in at runtime by the JavaScript.

	League name	name	nickname	shirtColor	
Edit	Premiership	Manchester City	Citizens	blue	Delete
Edit	Premiership	Manchester United	Red Devils	red	Delete
Edit	Premiership	Arsenal	Gunners	red	Delete
Edit	Premiership	Tottenham Hotspur	Spurs	white	Delete

New View All Delete All

```
<table id="listTable">
  <tbody></tbody>
</table>
<p>
<button type="button" onclick="gotoNew();">New</button>
<button type="button" onclick="gotoList();">View All</button>
<button type="button" onclick="dropDBTable();">Delete All</button>
</p>
```

3.5 INCLUDE COMMON SCRIPT CODE

Next in the HTML file comes a series of <script> statements to include the JavaScript files mentioned above. Since their contents are the same for all pages in these web applications, they are saved once and included here by reference with the src= attribute of the script tag.

```
<script type="text/javascript" src="sample.js"></script>
<script type="text/javascript" src="globals.js"></script>
<script type="text/javascript" src="entry_common.js"></script>
<script type="text/javascript" src="entry_form.js"></script>
```

If you would prefer to see their contents inline, edit the Web Application generator, change the line near the top to the following:

```
$inline='yes'
```

When you run the generator again, _includeScript will include the script contents inline.

3.6 META-DATA AND SCRIPT CODE FOR THIS PAGE

The rest of the HTML file is a long section of JavaScript that checks for page arguments, initializes some global variables with the meta-data for this page and its fields, and calls the

initialization code defined in the included JavaScript files. `_EntryJSparams` generates some code to parse the URL and set rowID, which records which particular Team is being edited. `_EntryJSvariable` creates the meta-data for the database schema and UI, based on the Attributes and relationships in the model; its output has already been discussed in the earlier sections.

At the end of the script we call the `init()` function, defined in `entry_common.js`. This will open the database for this application (Football), and create or update the table definition for this page (Team). Updating the table definition here allows us to add new Attributes to the model and regenerate, then simply refresh the page to see the changes. The init() function next initializes the display. It calls `displayRecentRows()`, defined in the same file, to fill in the table of recently edited entries defined in 3.4 above. If this page was loaded with a rowID parameter in its URL, `init()` will also fetch the values of the selected Team into the fields in the form (3.1 and 3.2 above), ready for editing.

Finally, we call the `initForeign()` function, defined in `entry_form.js`; the same function is also defined as empty for the main home page and list pages, as they have no foreign form fields to update. `initForeign()` populates the options list for foreign columns referring to other entries, e.g. the League field in the Team form in 3.1 above. It also calls `displayForeignRows()`, defined in the same file, to fill in the tables for references to this entry from other entries, e.g. the list of Players in 3.3 above.

4 Conclusion

In this example, we have demonstrated how domain-specific generators can help even a general purpose modeling language. Of course, to take things further the next step would be to create a proper domain-specific modeling language, taking into account the kinds of web applications you want to build. Many people have built such modeling languages, which can be divided into three categories:

1) for the in-house use of the organization that created them;

2) available as commercial tools, such as Mendix and OutSystems;

3) for anyone to use, such as WebML.

As we move down the list above, the languages try to reach a broader market and thus become less domain-specific – and hence cannot be as tight a fit with your own requirements on the kinds of applications you want or the kind of code you want. Conversely, the less experience you have with web applications, the more you are likely to accept someone else's "one size fits all" solution.

In a slightly different approach, the databases can be specified in online forms or wizards rather than a graphical modeling language, and the resulting application can be automatically hosted online, insulating you still further from technical issues: examples include Caspio and the sadly demised DabbleDB.

The defining question for your choice of technology is this: how much do you know about building web applications? The less you know, the more likely you are to be happy that some tool has already made many decisions for you. Conversely, the more you know, the less likely you are to be satisfied with the one-size-fits-all solutions of existing tools – or then if the existing tool offers many solutions, the more frustrated you are likely to be with having to fill in a myriad of forms and wizards, when you feel you could achieve the same results as quickly by hand coding. If you are building many web applications, the tools would force you to jump through the same hoops again for each application: at least with hand coding you could reduce the work by abstracting some common parts of your solution into frameworks.

If the choice was just between off the shelf tools and hand coding, most people who were capable would code their own web applications. That indeed is what generally happens today. With domain-specific modeling tools, there is a third choice: abstract the common parts still further. The parts that describe what kinds of applications you want to build go into the modeling language, your framework becomes a domain framework, and the instructions for how to use the framework are built into the generator. Now, you can build web applications of exactly the kind that you want, faster than in any of the previous ways. You can also give the modeling language to other people, who might be skilled at designing web applications but not at implementing them. In all likelihood, those people will prefer a modeling language other than UML.