Template Transformations in AutoMed

AutoMed Technical Report 25, Version 2

Charalambos Lazanitis
cl201@doc.ic.ac.uk

Minor Additions and Changes by
Nikolaos Rizopoulos
nr600@doc.ic.ac.uk

Table Of Contents

1. Introduction 3

Overview Of the Document 4

2. Using the Template GUI Tool 5

3. Guide to the Template API 11

3.1 The first API 11

3.1.1 Overview 11

3.1.2 An Example Template Definition 12

3.2 The Second API: The First Level Of Abstraction 20

3.2.1 Overview 20

3.2.2 Types 21

3.2.3 An Example Template 22

3.3 A Template Programming Language 24

3.3.1 The TDL Programming Language 24

Lesson 0: A Quick Syntax Index 25

Lesson 1: Getting Started 27

Lesson 2: Types 30

Lesson 3: ADD statement 31

Lesson 4: Template Inputs 36

Lesson 5: DELETE 40

Lesson 6: RENAME 41

Lesson 7: String manipulation in TDL 42

Lesson 8: Looping in TDL, the FOREACH statement 46

Lesson 9: Manipulating the Scheme of Objects 50

3.3.2 Runtime Template Transformations 54

1. Introduction

During the Database Schema Integration process, it is known that in practice the sequence of transformations that have to take place in order to go from the initial Schema to the target Schema is not completely random. Transformations are being performed to resolve conflicts, and of course each conflict requires a sequence of transformations. It becomes apparent therefore, that if we could define the sequence of transformations that are required to resolve each particular conflict in a generic manner, we could reach our target schema a conflict at a time rather than a primitive transformation at a time.

This is partly what the tool developed by Nikolaos Rizopoulos attempts to achieve. A sequence of transformations is called a template. It is defined in a schema independent way, and defines as variables what is dependent on the schema, which have to be filled in by the user to form an instance of the template. As soon as the user defines the required variables, the sequence of transformations can be executed on the source schema. The tool is model independent, but each template is dependent on a particular model. A basic understanding of the functionality of this tool is required in order to follow this document. A summary of the minimum required things you need to know will be given here, but if you want more information on this tool you can find it at:
http://www.doc.ic.ac.uk/automed/publications/Riz01.ps.gz

An attempt has been made to create an API that will wrap the NR tool, both in the way that the templates are defined and the way they are used.

Although this API does not add any extra functionality to the existing tool, it offers a much simpler way of performing what up to now were hard and tedious tasks. Taking a familiar analogy, it is like using a high level programming language to wrap an assembly set. So although the high level language does not add extra functionality to the assembly set, it facilitates writing complex programs and arguing about their correctness. Furhtermore, languages often try to guarantee the correctness of a program up to some level, by introducing types, variable scoping and other features that check part of the logic of the program in compile time.

In fact the above analogy was the guideline to the development of the API. Previously, defining a template seemed more or less like filling up a number of tables in a relational database, through a low level API. Now, in the first level abstraction, a template definition is seen as a sequence of statements inside, each having arguments and return values. For this API, types have been introduced for each argument and each return value. This first level API, has been further wrapped by a second level API, that simulates the definition of a template as writing a program in a simple procedural language. The notion of scopes, and checking of the logical order of variables have been introduced here. This is what we called TemplateCompilerSimulator, and it will be explained quite thoroughly in this document.

Overview Of the Document

Section 2 describes how to use a Graphical Tool developed to implement template transformations on schemas that are defined using the templates API.

Section 3 describes how the API for defining templates has been created and how it was evolved

> **Section 3.1** describes the tool developed by Nikos Rizopoulos and it is the lowest and basic level of the entire API
>
> **Section 3.2** describes the first level of abstraction on top of the basic API
>
> **Section 3.3** describes the syntax and semantics of a pseudo-programming language developed on top of the first level of abstraction API, which tries to simulate a real programming language that is used to define templates.

2. Using the Template GUI Tool

This section explains how to use the GUI developed to perform template (composite) transformation on schemas. We are assuming that you have the Automed API already installed on your machine along with the necessary resources.

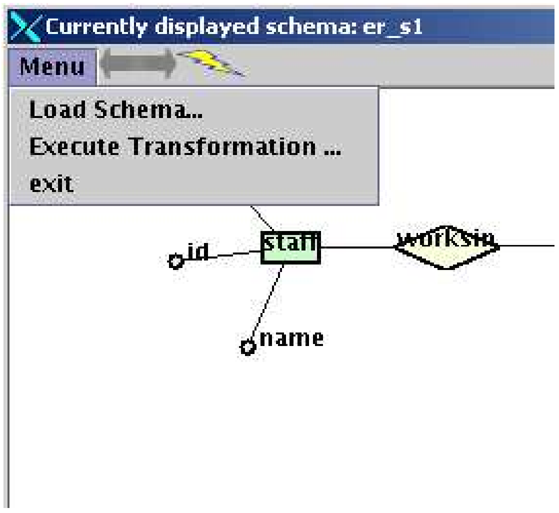
First of all to define the demonstration templates go the **examples** directory and simply type:

```
javaenv.sh make templates
```

There is a demonstration GUI that displays schemas and is able to pop-up the dialogue that implements the templates that have already been defined. This can be run from the examples directory, by typing:

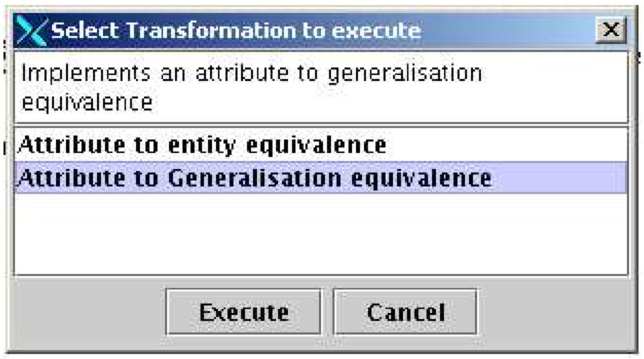
```
javaenv.sh make templatesgui
```

This will pop-up a window and a dialog that will prompt you to select a **schema** to load. After doing that you will see something very similar to what is shown bellow:



From there, there is only a handful of options you can do, like **loading** an other schema, or moving the objects in the display around. By far though, the most ‘exciting’ option you have is to chose a **composite transformation** to execute. To emphasize the excitement of performing this action we have offered two alternative ways of performing it. That is choosing it from the menu, or clicking on the lighting image on the menu bar.

After doing that, a dialog box that looks like the following will pop-up:


Select Transformation to execute

Implements an attribute to generalisation equivalence

Attribute to entity equivalence

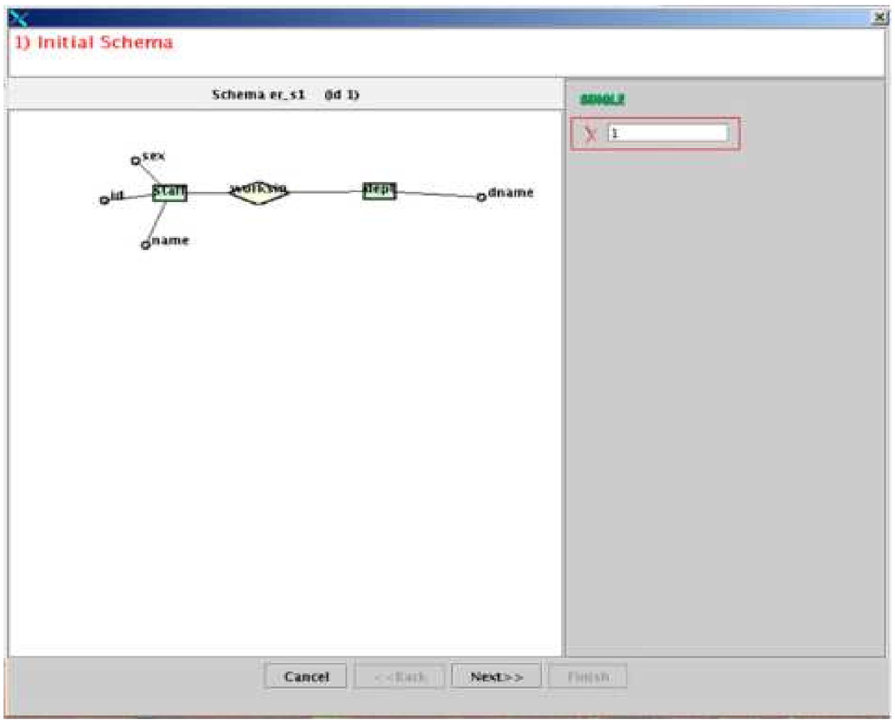
Attribute to Generalisation equivalence

Execute Cancel

This gives you a list of all the transformations you can execute, and on selecting any of them you get its description displayed at the header. If you are not satisfied with what you see, you can always ‘**Cancel’**, but I am sure you will be eager to click on ‘**Execute’**. This will cause yet an other Dialog Box to pop-up (the most interesting one so far) that will let you implement the Transformation you chose. So, let’s say that we chose ‘**Attribute to Generalisation Equivalence’**. Then we’ll get a dialog that will let us implement the Attribute to Generalisation Equivalence transformation (no surprises there).

Let’s go step by step, filling up the dialog box, and demonstrate the features that it provides.

Almost always the first argument of the dialogue will be the initial schema. (In fact if you use the **TemplateCompilerSimulator** (Section 3.3) to define your templates, this will always be the case.) The window will look like the following:


1) Initial Schema

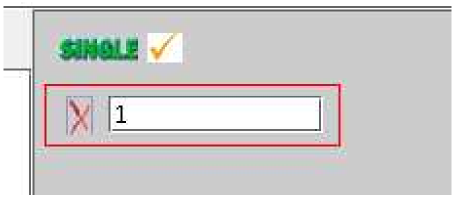
Schema er_s1 (Id 1)

1

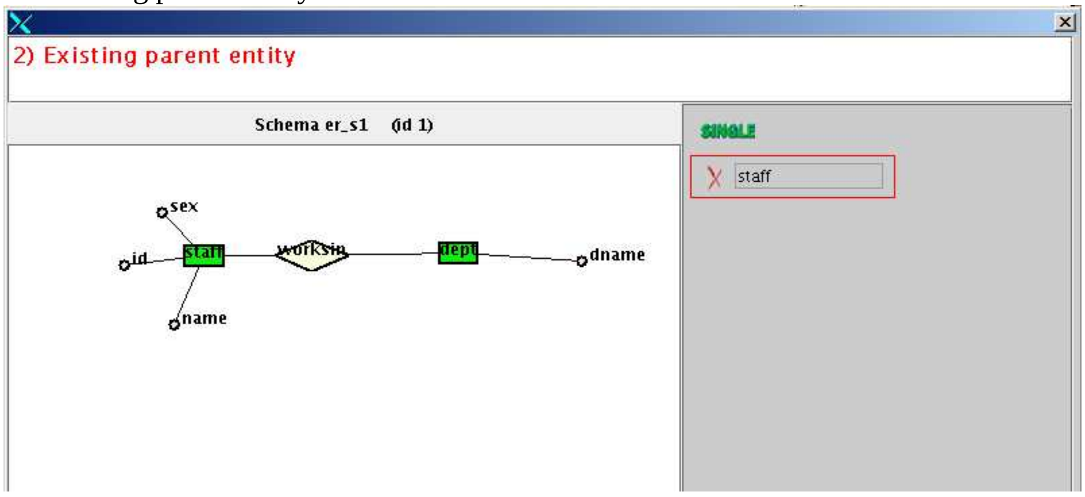
The things to notice:

- The top panel that gives the description of the **current argument**
- The display of the schema that the transformation is done upon,
- An input panel on the right that gets the inputs for the arguments.
- The "**SINGLE**" sign that denotes that the argument is a single argument (in contrast with a **LIST** argument).

An argument of type SCHEMA, is by default set as the currently displayed schema. If for any reason you think that the value should be something else, you can edit the box and type anything you want. If you click NEXT, it will try to SET the argument (The argument will be set only if the value is valid). If you want to check if it is valid before moving on, you can click on the SINGLE icon, and if it is successfully set, you will get a tick as an indication:



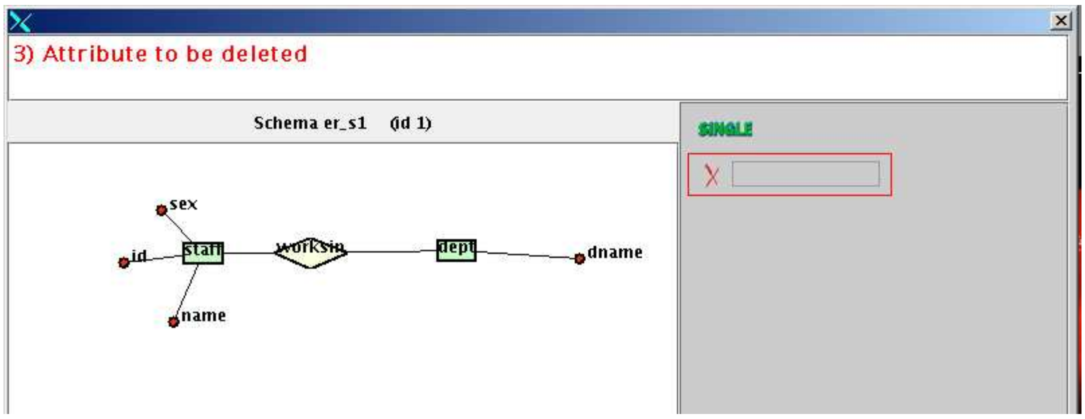
So now we are ready to click **next**, and go on to the next argument, which happens to be the existing parent entity:


2) Existing parent entity

Schema er_s1 (id 1)

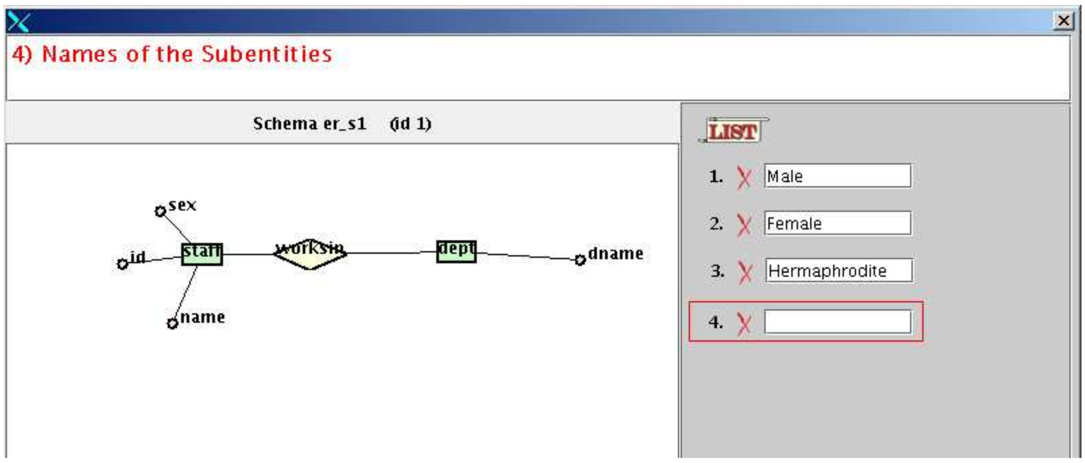
Since, while defining the template, we specified that this argument should be an **entity**, all the entities are highlighted, and are accepting mouse events. Clicking on staff will pass its value to the box to the next. Note that since we are looking for an OBJECT, the input box has been disabled, and thus we can only enter values by clicking on the appropriate objects in the schema display (You cannot edit values from the keyboard)

click NEXT:


3) Attribute to be deleted

Schema er_s1 (id 1)

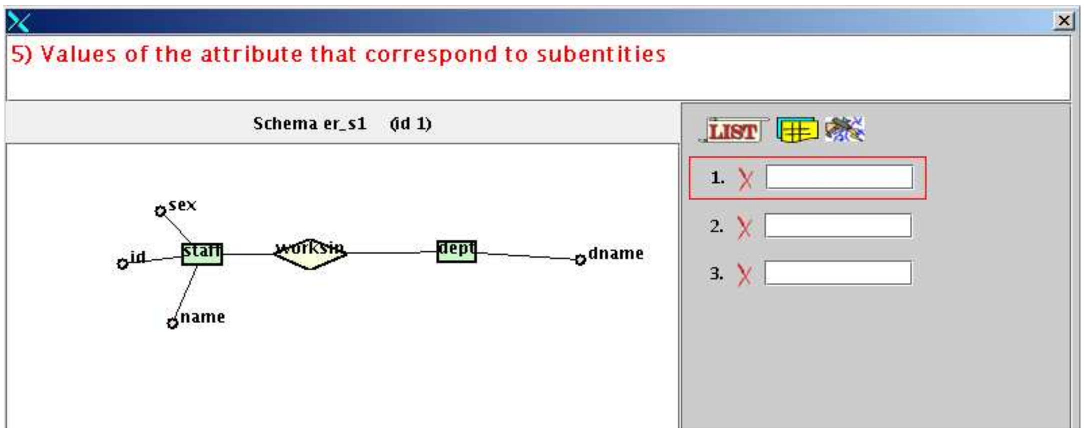
Similar story as above, just this time the attributes are highlighted instead. We chose SEX, (we are implementing the sex to Male-Female transformation), and click NEXT.


4) Names of the Subentities

Schema er_s1 (id 1)

1. Male
2. Female
3. Hermaphrodite
4.

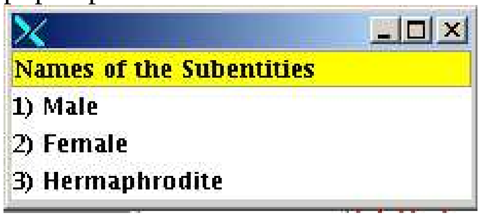
This asks for the names of sub-entities. Unlike the previous cases, now the argument is of type list, and you can enter as many elements as you like. We chose to enter the three genders found in nature, and click NEXT.


5) Values of the attribute that correspond to subentities

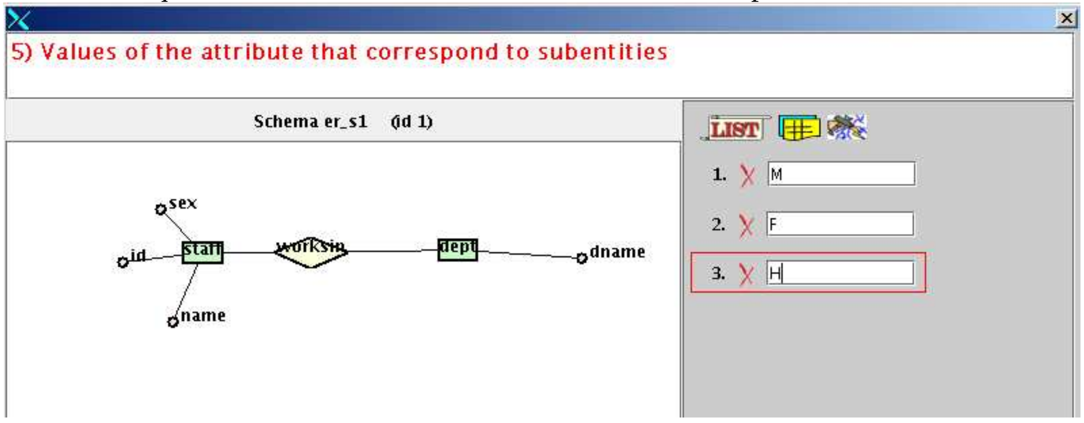
Schema er_s1 (id 1)

This asks for the values that correspond to the sub-entities defined previously. Apparently when the template was being defined, this list had a reference to the previous list , which means that the two lists should be of the same size. This had as a result the list to be set, fixed to 3 elements, and was mad rigid, i.e. not expandable. To demonstrate some of the features provided think of the following scenario:

The user forgets in which order the subentities where layed out for the previous argument (i.e. male before female or female before male?). After a few moments of panic, she wanders what the yellow button above does, and decides to click on it. (of course she could just have clicked BACK and see what the input was but this would ruin our story). Anyway, she clicks the yellow button, and to her amazement, the following window pops-up:


Names of the Subentities
1) Male
2) Female
3) Hermaphrodite

This window describes the way all the referenced lists have been layed down. So she realises that Male comes before Female (as always J), and she can now go on filling up the list as required: M for Male, F for Female, H for Hermaphrodite.


5) Values of the attribute that correspond to subentities

Schema er_s1 (id 1)

1. M
2. F
3. H

But now she suddenly realised that the last Tapeworm has left the college 10 years ago, and there is a new non-Hermaphrodite-hiring policy. It therefore becomes apparent that the Hermaphrodite subentity has become obsolete. She tries to click on the “X” button next to the 3rd element but the element is not removed. (What is written there is deleted but the input item remains). She clicks on the “X” button 35 more times, turns the screen on and off a couple of times, but the input item is still there. She then realises that the list size has been FIXED. How can I ‘break’ it, she asks her self? This is when the word ‘break’ has given her a hint of the use of the icon showing a hammer breaking a glass. Clicking that, makes the list expandable again. She can thus remove the third item, and clicks back to fix the previous list as well.

Note that since an input item has been removed, clicking **back**, the previous list will be fixed to two elements. The API has been structured such that if we have a set of referenced lists, all the lists are fixed to the number of elements that the last list in the set has been set to.

Clicking NEXT, will ask for the generalisation name, and since this is the last argument, the FINISH button will become enabled.

If all arguments are set correctly the template will be executed, otherwise you will get an error message and directed to the argument that gave the error. In the former case, window will close and the new schema will be loaded to the displaying frame.

3. Guide to the Template API

In the introduction we said that all you have to know about the NR tool can be found on the technical report for the tool. Well, this is not completely true. Although that documented the tool at that time thoroughly, some extra features have been added to increase the functionality. What was added, is basically the ability to dynamically define strings with the DEFINE_DYNAMIC_STRING_STATEMENT, the use of the RENAME_STATEMENT, a statement that accesses the scheme of an object and one for accessing a particular element in a list. Furthermore, it has been upgraded such that it uses the **REPS** API instead of the **STR** API to perform the transformations. (If the words REPS and STR sound Greek to you, now it's a good time to visit this link: http://www.doc.ic.ac.uk/automed/techreports/index.html where you can find all the technical reports you need about AutoMed project. The AutoMed website is at: http://www.doc.ic.ac.uk/automed).

So, after these adjustments to the initial state of the NR tool, it now looks as described below:

3.1 The first API

3.1.1 Overview

It consists of 7 statements:

1) NEW_OBJECT
2) PRIMITIVE_TRANSFORMATION
3) FOREACH
4) INVOCATION
5) DEFINE_DYNAMIC_STRING
6) SCHEME_LIST
7) INDEX_LIST_STATEMENT

(The last 3 were added later)
The Primitive transformation statement can be parameterized, such that it performs the following statements:
ADD, DELETE, RENAME, CONTRACT, EXTEND

Each statement contains several **arguments**. Some of them store variables that are used to define the statement, and some are defined after the statement has been executed to define the results of the statement. A template consists of a number of arguments that define the following:

1. The template's inputs (that will eventually be defined by the user)

2. The template's outputs
3. A sequence of statements

On the lowest level, a template is really a set of interconnected tables in a relational database. At this time, defining a template is not far from filling up these tables. Although there is some API that somewhat abstracts away from the actual tables, in order to define a template you will have to be very familiar with the way the statements are defined in the table level, and even so, defining a complex template can be extremely complicated.

3.1.2 An Example Template Definition

In order to get a view of how a template is defined, I will try to summarize the required steps in order to define a standard but fairly complex transformation in the sense that it uses most of the features of the tool. The transformation is the **Attribute to Generalization Equivalence.** (If you are not sure what this transformation is then you probably shouldn't be reading this document). Note that we won't bother to define functions and constraints, as this will add too much complexity.

If we define this transformation in a generic level, then we define it as:

- Starts on an initial schema **Si**
- Adds a number of subentities **(e1…en)**
- Adds a generalization **G** from a parent entity **E** to the set of subentities
- Deletes the general attribute **A**

So the first thing to be done is to ask the user for the variables in the above list:

1. An initial schema ID (Si),
2. A List of subentity names (e1…en)
3. A generalization name (G)
4. The ID of the existing parent entity
5. The ID of the general attribute

These five input arguments have to be defined as records in a table, with a **trans_id** and an **arg_pos** which are the primary keys of each argument. The trans_id is taken care by the existing API, but the argument position for each argument has to be defined explicity. A **description** has to be provided for each argument and a Boolean whether the argument is a **return** argument or not (**input** arguments are not return arguments). The **type** of each argument (whether it is a list or an id or a name etc) is only revealed to the user by the argument description and there is nothing to enforce it.

A possible instance of the above would be the table that follows. (Assume that the trans_id happens to be number 18)

Trans_id	Arg_pos	Description	isReturn
18	1	Initial Schema Id	0
18	2	Subentity names	0
18	3	Generalization name	0
18	4	Id of existing parent entity	0
18	5	Id of existing general attribute	0

(If you read the documentation of this tool you might realize the existence of two more rows that define a table and a field, but they are neglected here)

Omitting several details, the next step is to define the first statement in the template. The first statement is the one that creates all the subentities. Their number is not fixed, and is defined by the number of names given for the second input by the user. The subentitties are created inside a loop, which is a **foreach** statement. Unfortunately the foreach statement is the most complex one. Basically it works similar to a foreach statement in a real programming language (if you are familiar with shell scripting the similarity is greater). It itterating a list and executes a particular block, once for each element in the list. The special feature in this case is that **<u>the foreach block is a single template</u>**. It can be any template, perhaps a complex one that can be defined in a different file and could have a function by itself. In our case however, the template is fairly simple. It just adds a single entity. Even so however, will treat this template as a separate entity, since inspite its simplicity, it demonstrates the main features of any template.

<u>The foreach block template:</u>

Again, we should first define the **inputs** of the template, which is the name of the entity and the initial schema. Something that I haven't explicitly mentioned (but implied) is that in a template we can also define **output** arguments. That is arguments that are unknown before the execution of the template, and are set by the template execution. This feature might seem not so useful for a main template, but it is particularly useful for a foreach template (since the purpose of the loop is to produce a number of items that are going to be used later). In this case our output should be the ID of the created entity. So, by defining the template inputs as described in the case of the external template, we get something like:

Trans_id	Arg_pos	Description	isReturn
19	1	Initial Schema Id	0
19	2	Entity name	0
19	3	Created entity Id	1

All the template does, is to add a single entity to the given schema. So, you would think that it only requires a single statement, but you would be wrong. It actually requires two. A statement to create the new object and one to add it to the schema.

The presence of two statements rather than one was necessary since when the tool was initially developed, the underlying API (STR) required an object to be first explicitly created and then added to the schema. When the tool was upgraded to the REPS API, it appeared that it was possible to merge the two statements to one. However, for reasons of backwards compatibility and simplicity of the upgrade, it was decided to keep both statements as before. (At this point I could tell you that after the upgrade, the new_object statement only stores some values temporarily and does not really change anything in the database, but I won't, since this should not concern you as a user of the tool).

So, let's start with the new_object statement. The statement fingerprints are as follows:

Arg_pos	**Description**
1	Construct ID
2*	Schema ID
3*	Created object ID
4	Value (name of object)
5	IsTemp
6	Scheme ID (arg_id of object scheme)

The arguments that take an (*) are passed by reference and not by value. In simple terms, at those positions, the arguments are not known before the execution of the statement, but and are set by the statement. Their value can therefore be obtained after the statement is executed, by referring to their position in the statement arguments.

In order to define this statement, we must first define a new TemplateArgumentSequence. (call to the constructor of the `TemplateArgumentSequence` class). This will dynamically create a new arg_id, in the table, that will identify the arguments of this statement. The arguments that are passed by reference (output) should not be defined. The arguments that are passed by value (input), can either be:

- entered directly OR
- point to an other statement argument OR
- point to one of the arguments of the enclosing template

Here's the code that creates a new entity:

```
//Create a new argument sequence
TemplateArgumentSequence newEntityArgs = new TemplateArgumentSequence();

//CID is the ID of entity
newEntityArgs.createArgument(CompositeTransformation.ARG_CID_IN_NEWOBJECT,
String.valueOf(erEntity.getCID()));

//Enitity name is the second input argument
newEntityArgs.createArgument(CompositeTransformation.ARG_VALUE_IN_NEWOBJECT, 2);

//is temp is 0 (not a temporary)
newEntityArgs.createArgument(CompositeTransformation.ARG_ISTEMP_IN_NEWOBJECT, "0");
```

```
//--------------------------------------------------------------
//DEFINE THE SCHEME ARGUMENTS (list that consists of the entity name)
TemplateArgumentSequence newEntityScheme = new TemplateArgumentSequence();

//The first and only element in the scheme consists of the object id of the created entity
//that will be set after the statement is executed. It will be stored in the 3rd argument position
newEntityScheme.createArgument(1, newEntityArgs, CompositeTransformation.ARG_OBID_IN_NEWOBJECT);
//--------------------------------------------------------------

//The scheme points to the above template argument sequence that defines the scheme
newEntityArgs.createArgument(CompositeTransformation.ARG_SCHEME_IN_NEWOBJECT,
String.valueOf(newEntityScheme.getSequenceId()));
```

As you might or might not have noticed, you define an argument by invoking a `createArgument` method on a `TemplateSequenceArgument` object. Extracting lines of code we demonstrate how we can define the three types of arguments stated above:

- Direct:

```
newEntityArgs.createArgument(CompositeTransformation.ARG_ISTEMP_IN_NEWOBJECT, "0");
```

- Point to an other statement argument:

```
newEntityArgs.createArgument(CompositeTransformation.ARG_SCHEME_IN_NEWOBJECT,
String.valueOf(newEntityScheme.getSequenceId()));
```

- point to one of the arguments of the enclosing template

```
newEntityArgs.createArgument(CompositeTransformation.ARG_VALUE_IN_NEWOBJECT, 2);
```

Note that the scheme of an entity only consists of the entity name. This is not generally the case. If we had an attribute for example the scheme would consist of the attribute name and a pointer of where the parent attribute sits on the database.

The next thing to do is to add the created entity to the schema. This is a **primitive transformation** statement, which has to be parameterized to be an **add statement**:

```
//create new argument sequence
TemplateArgumentSequence addNewAttributeArgs = new TemplateArgumentSequence();
//Primitive transformation is ADD
addNewAttributeArgs.createArgument(CompositeTransformation.ARG_ACTION_IN_PRIMITIVE, "add");
//The initial object is not applicable so its value is set to -1
addNewAttributeArgs.createArgument(CompositeTransformation.ARG_FROMOBJ_IN_PRIMITIVE, "-1");
//The object that will be added is the one we just created in the above statement
addNewAttributeArgs.createArgument(CompositeTransformation.ARG_TOOBJ_IN_PRIMITIVE, newAttributeArgs,
                        CompositeTransformation.ARG_OBID_IN_NEWOBJECT);
//Initial schema is the first argument of the template
addNewAttributeArgs.createArgument(CompositeTransformation.ARG_FROMSCHEMA_IN_PRIMITIVE, 1);
```

```
//The final schema is the the final schema in the previous transformation
addNewAttributeArgs.createArgument(CompositeTransformation.ARG_TOSCHEMA_IN_PRIMITIVE,
                                   newAttributeArgs, CompositeTransformation.ARG_SID_IN_NEWOBJECT);
//We could define the function here but we are not bothering at this document
addNewAttributeArgs.createArgument(CompositeTransformation.ARG_FUNCTION_IN_PRIMITIVE, "");
//We could add constranits here but we are not bothering for this document
addNewAttributeArgs.createArgument(CompositeTransformation.ARG_CONSTRAINTS_IN_PRIMITIVE, "");
```

So now that all the statements of the template have been defined, it is time to define the output arguments of the template. Here's the code that does that:

```
 //Template argument sequence to hold return arguments
TemplateArgumentSequence returnedArgs = new TemplateArgumentSequence();
returnedArgs.createArgument(4, newAttributeArgs, CompositeTransformation.ARG_OBID_IN_NEWOBJECT);
returnedArgs.createArgument(5, addNewAttributeArgs,
                                        CompositeTransformation.ARG_TOSCHEMA_IN_PRIMITIVE);
addAttribute.createReturnedArgumentsDefinition(returnedArgs);
```

This says that the template has two return arguments, one defined as the 4th argument of the template and is the id of the created object, and the second one is the 5th argument of the template and is the final schema.

Now that we created the template, we need to create the execution for it. Although this might seem quite unintuitive at first, it is actually a rather crucial aspect. What we have done up to now is created some "*methods*" that can be called. Actually we didn't even do that. We created the arguments for some methods that can be called. We now need to say what methods each of those sets of arguments correspond to, and the order by which they will be called.

Here's the code:

```
addAttribute.createStatementExecution(1,
                  Statement.getStatement(CompositeTransformation.NEW_OBJECT_STATEMENT),
                  newAttributeArgs);

addAttribute.createStatementExecution(2,
                Statement.getStatement(CompositeTransformation.PRIMITIVE_TRANSFORMATION_STATEMENT),
                addNewAttributeArgs);
```

A Small Discussion

Now a small discussion that will be an appetizer of what will follow in the next sections. You have witnessed how a really simple template can be defined. Actually it is a simplified version of a really simple template.
You should have noted two things (at least)

1) Firstly that there is some equivalence of all of this code to a program written to some (procedural) language. All it does it stores some arguments in some database, and provides some sort of guidance of how these arguments will be used to execute a program that does something. It also has the notion of input and output.
2) The second thing to notice is how difficult it is to realize the first point.

If you select at random a computer science undergraduate, the chances is that he or she has been told the following rule:
“Any problem in computing can be solved by adding an extra layer of abstraction.”

Using this rule combined with some common sense and observing points 1 and 2 will probably make you realize that an abstraction of this code might very well lead us into creating a programming language.

Other things you might have noticed is that there is nothing that prevents you from(or at least helps you avoid) writing code that makes no logical or even implementational sense. There is also nothing that can prevent the user from inputting invalid input, or help the program realize that the input that was entered was invalid.
One might thing that introducing some sort of typing (perhaps at a higher level) might solve the problem.

This concluded the brief discussion and I hope you understood the need of providing a wrapper to this API and might have got a hint of how to go about developing one.

OK, back to the code.

Now that we defined the foreach template, we can define the foreach statement as well, and link the template to it. There are several details here that mainly have implementational importance and I will avoid stating them. If you wish to learn exactly how this statement works you should refer to Nikos Rizopoulos thesis. I will try instead to give you a simplified version of the truth, hiding much of the implementational details.

The foreach statement is defined to have the following paramenters:

Arg_pos	**Description**
1	Intital schema
2	sequences
3	singletons
4	template to be executed
5*	Return value #1
6*	Return value #2
7*	Return value #3 (and so on)

I have already given you a taste of how a template and a statement is defined, so from now on I will try to be more descriptive rather than code-dumping.

Before I try to explain what each argument means, let’s recall what this statement is supposed to do.
It should go through a list of names given as input to the (external) template (at position 2) and for each of those names it should add an entity with the particular name.

We already defined the template that adds the entity. A high level definition of this template we defined is:

```
New_entity_ID = addEntityTemplate(Entity_name, Initial_Schema);
```

So the foreach statement should do something like:

```
foreach[name IN input_name_list]
{
      addEntityTemplate(name, current_schema);
}
```

It would be useful to be able to accumulate the entity ids created so that we could accumulate them later. ie do something like this.

```
foreach[name IN input_name_list]
{
      ID = addEntityTemplate(name. currentSchema);
      resultList.add(ID);
}foreach_return = result_list;
```

A more general definition of the statement would be:

```
foreach[arg1 IN list1,  arg2 IN list2, …,argN IN listN]
{
      (res1, res2, …,resM) = performTemplate(template args);
      resultList1.add(res1);
      resultList2.add(res2);
      …
      resultListM.add(resM);

}foreach_return1 = resultList1
 foreach_return2 = resultList2
      …
 foreach_retrunM = resultListM
```

Note that all iterated list should have the same number of elements (K) and the loop will be iterated K times as well.

So here’s what each of the arguments state:

1. **initial schema**: The schema on which the foreach statement starts on. This is updated every time the loop is iterated. So you can get the final schema of the statement from the same position.

2. **sequences:** A list of lists that define the iterated listst inside the template. Taking the abstraction above, it would be a pointer to a list containinng the lists: list1, list2 … listN.
3. **Singletons**: inside the loop there might be some loop invariant variables. So singletons points to a list of loop invariants.
4. **Template to be executed** should somehow point to the template to be executed inside the loop. We are ommitting the technique used to pass parameters to the template. You can find this technique on the NR thesis.
5. **The return values** (returnList1 .. returnListN are pointed at by arguments 5 onwards. So resultList K can be found at position 4+K.

For our case there are no singletons, and only 1 return argument.

The next statement is to add a generalization. The techniques used are the same as when we added an entity (but quite more complex), so the idea is the same. The complexity of this statement comes from the fact that the scheme of the generalization is quite more complex than the scheme of the entity.
It should contain:

1. the name of the generalization, which can be taken from the third input argument of the template.
2. the parent entity id, which can be taken from the 4th input argument.
3. a list with the subentities. This list is the return argument of the foreach statement we just created. so we can get it from there.

This means that the format of this statement will be similar to the one for adding an entity. This time however the scheme arguments should contain a reference to the first input argument at position 1, a referce to the 4th input argument at position 2 and a pointer to the 5th (first return) argument at position 3.

If you look at NR thesis you will realise that adding the subentities to the scheme required a new statement called EXTEND_GENERALISATION_SCHEME. During the upgrade of the tool, it became apparent that this statement could be avoided.

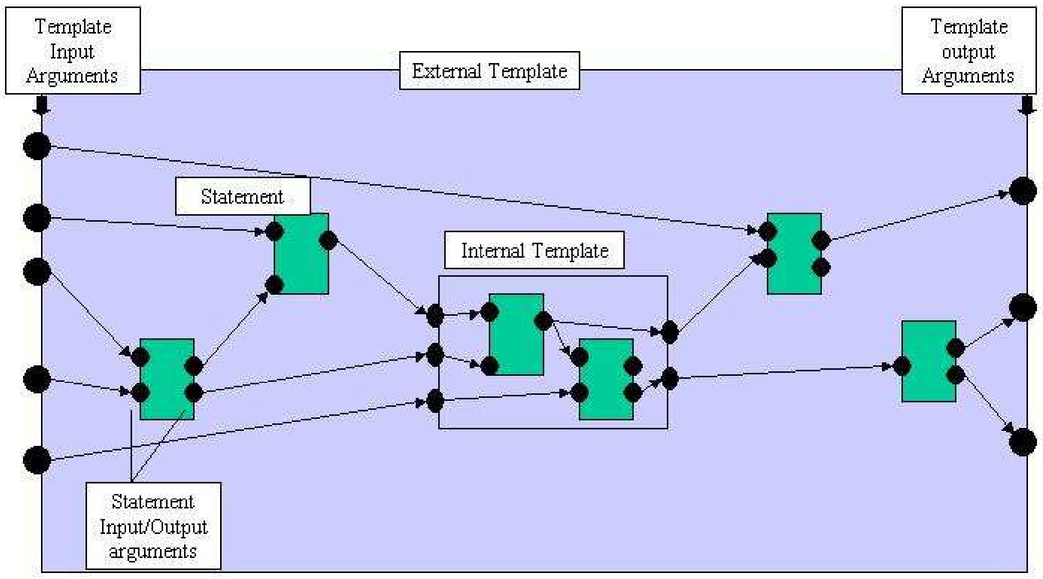
Hopefully by now you have realized the purpose of generic transformation called templates and the way they are defined using the initial API. Furthermore we demonstrated that this API is functional but sometimes tedious and unintuitive which brings the need of a higher level API that will abstract the current one.

3.2 The Second API: The First Level Of Abstraction

3.2.1 Overview

The first layer over the existing API aims to abstract the following things:

1. Prevent either the programmer (who defines the template) or the user (who implements the template) from using **invalid input**. For example you prevent the user of inserting a string input where a number is expected.
2. Hide arguments from statements. Looks at the arguments of a statement as **input** arguments and **output** arguments in which case you only define the input arguments when you define the statement and only see its output arguments when you want to use it.
3. Give a **graph abstraction** to the program.
4. Avoid a lot of the **initialization** and **finalization** code which is standard for all the templates



Abstraction Of Template Definition As Graph

In this API, templates and statements are treated as objects. All statements are derived from the **`TStatement`** object, (examples are `AddStatement`, `DeleteStatement`, `CreateListStatement` etc). Each template is an instance of the **`TTemplate`** object. Each statement has **`TStatementArguments`** and each template has **`TTemplateArguments`**. Both types of arguments are derived from the **`TArgument`** class. Statement input arguments only have setter methods while output arguments only have getter methods.

For example an AddStatement has **setters** for:

```
setFromSchema // initial schema
setFunction // sets the function
setConstraints
setConstruct // sets the construct of the object to be added
setValue // sets the value of the new object (ie name)
```

and **getters** for:

```
resultingSchema  // The argument for the resulting schema after
                  //execution of statement
resultingObject  //The  argument of the object id of the created object
```

Note that all getters return `TStatement` arguments. Setters for arguments which can be passed by reference can be set by a `TArgument`..

3.2.2 Types

Now arguments have **types**. This guarantees that you do not pass arguments around that do not make sense. For example, you are not allow to refer to an argument that defines a construct in order to define a schema. Here's a view of the type tree in the language.

- class uk.ac.ic.doc.automed.templates.wrapper.TType
  - class uk.ac.ic.doc.automed.templates.wrapper.**TType.List**
  - class uk.ac.ic.doc.automed.templates.wrapper.**TType.Object**
  - class uk.ac.ic.doc.automed.templates.wrapper.**TType.Schema**
  - class uk.ac.ic.doc.automed.templates.wrapper.**TType.Secret**
  - class uk.ac.ic.doc.automed.templates.wrapper.**TType.String**
    - class uk.ac.ic.doc.automed.templates.wrapper.**TType.Constraints**
    - class uk.ac.ic.doc.automed.templates.wrapper.**TType.Function**
    - class uk.ac.ic.doc.automed.templates.wrapper.**TType.Name**
  - class uk.ac.ic.doc.automed.templates.wrapper.**TType.Supertype**

Note that `TType.Secret` is for arguments that are not passed around (for example there is an argument that defines whether a particular object is temporary or not). `TType.SuperType` was introduced later when we needed a super type for lists whose type whose element type was unknown (see section 3.3).

3.2.3 An Example Template

Here's an example code using this API which performs the same template transformation as the example given previously, that is attribute to generalization equivalence.

```
        public static void defineTemplate8()
                throws Exception
        {
 //Define the constructs and models//

                System.out.println("Mandatory Attribute to Generalisation");
                Model er = Model.getModel("erOld");
                Construct entity = er.getConstruct("entityOld");
                Construct attribute = er.getConstruct("attributeOld");
                Construct relationship = er.getConstruct("relationshipOld");
                Construct generalisation = er.getConstruct("generalisationOld");

//Define the template inputs with types//

                TTemplate templ = new TTemplate("TEST 8: Mandatory Attribute to
Equivalence");

                templ.setInput("initSchema", "Initial Schema", TConstants.SCHEMA);
  //The above sets input argument called “initSchema” to be  of type SCHEMA, and
  //user is prompted to add it,  by the string: “Initial Schema”.

               templ.setInput("GenName", "Generalisation Name", TConstants.NAME);
                templ.setInput("entity", "Entity ID", entity);
                //templ.setInput("Attr", "Attribute ID", attribute);
                templ.setInput("entityNames", "Entity Names", TConstants.NAME,
TConstants.LIST);



                ForeachStatement foreach = new ForeachStatement();
                foreach.setSchema(templ.getInput("initSchema"));
                foreach.setIterator("name", templ.getInput("entityNames"));
                foreach.setVariable("schema", foreach.resultingSchema());

                        AddStatement addEntity = new AddStatement();
                        addEntity.setConstruct(entity);
                        addEntity.setValue(foreach.getCurrent("name"));
                        addEntity.setFromSchema(foreach.getCurrent("schema"));
                        addEntity.setScheme(new Object[]{addEntity.self});
                foreach.setBlock(new TStatement [] {addEntity});
                foreach.setOutputList("subentities", addEntity.resultingObject());


                AddStatement addGen = new AddStatement();
                addGen.setFromSchema(foreach.resultingSchema());
                addGen.setValue(templ.getInput("GenName"));
                addGen.setConstruct(generalisation);
                addGen.setScheme(new Object [] {addGen.self,
                                        "total",
                                        templ.getInput("entity"),
                                        foreach.getOutput("subentities")
                                       }
                        );






                templ.setOutput("finalSchema", "final schema", addGen.resultingSchema());

  //The following is very important//
                templ.setStatementSequence(new TStatement []{foreach, addGen});
                templ.execute();
        }
```

Some things to notice:

- `setSchema` is a setter for all statements.
- Notice how the `foreach` argument is constructed. A `foreach` statement iterates through a list. `foreach.setIterator(“name”, aListArg),` says that the statement iterates through the list pointed at by `aListArg`, and at each iteration the iterated element wll be refered to by: `foreach.getCurrent(“name”);` The `setVariable` function sets variables that can be used within the loop (not the iterated ones).
- Note the last two arguments of the template. The first one actually links the statements to produce a program. The last one is the statement that actually updates the database.

You might have realized that this API does not add any extra functionality to the previous one. It just sits above the existing API and provides the abstractions we have described. Remember that our main goal is to make something that looks like a programming language. This is not quite yet a programming language yet but you might be able to see how it can be used to create one.

The following section will describe a pseudo-programming language and its pseudo-compiler that was created on top of this API. Again this will not add any extra functionality only extra usability.

3.3 A Template Programming Language

Question: In which language is this program written?

```
START();

        FOREACH();  NAME subName =IN (subEntityNames);
                   NAME popName =IN (populationOfEntities);
                   OBJECT att =VARIES_WITH(existingAttribute);
                   OBJECT parent =VARIES_WITH(parentEntity);
                DO();


                        FUNCTION f1 = DEFINE_FUNCTION("@subname(X):-@att?scheme(X,@nameGiven)");

                        OBJECT newEntity = ADD(CONSTRUCT.IS(entity),
                                                SCHEME.IS(new Object [] {my(subName)}),
                                                FUNCTION.IS(f1)
                                               );
                        OBJECTLIST createdEntities =COLLECTS(newEntity);
        ENDFOREACH();

        ADD (generalisation,
             new Object [] {my(genName),
                                "total",
                                parentEntity,
                                createdEntities
                          }
        );
        existingAttribute.DELETE();
END();
```

Answer: Java.

If you failed to answer this question correctly then the design of the language was successful. The idea is to create a programming environment where the programmer writes **a java program** in such a way that it gives the illusion that he is writing some procedural language.

3.3.1 The TDL Programming Language

I have to admit this language is not yet to the state where it would deserve a name given to it. However for this document (and for this document only) we have to give it a name for convenience. After serious considerations (that lasted 5 seconds) I decided to name it **TDL**, which stands for Template Definition Language.

In this section you will learn how to write templates using this “language”, which has been built using the API described in the previous section. We will describe the language in several lessons each describing a different aspect of it.

Lesson 0: A Quick Syntax Index

If you already know much of the syntax of the language and only need to recall some of the statements and what they do you will probably appreciate this lesson. The following table prompts you to which lesson you should read for each of the language statements and keywords

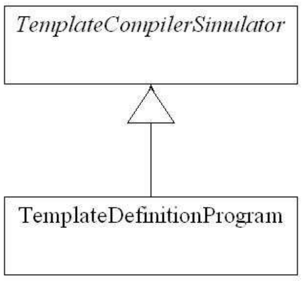
Statement/Keyword	Explained at lesson
`ADD`	3
`ALIAS`	7
`askForConstraints`	4
`askForConstraintsList`	4
`askForConstraintsList`	4
`askForFunction`	4
`askForFunctionList`	4
`askForFunctionList`	4
`askForName`	4
`askForNameList`	4
`askForObject`	4
`askForObjectList`	4
`COLLECTS`	8
`CONSTRAINTS.IS`	3
`CONSTRUCT.IS`	3
`CONTRACT`	5
`CREATE_LIST`	9
`defaultModel()`	1
`DEFINE_CONSTRAINTS`	7
`DEFINE_FUNCTION`	7
`DEFINE_NAME`	7
`DEFINE_STRING`	7
`defineTemplate()`	1
`DELETE`	5
`DO`	8
`ELEMENT_AT`	9
`END`	1
`ENDFOREACH`	8
`EXTEND`	3
`FOREACH`	8
`FUNCTION.IS`	3
`IN`	8
`INPUTS`	4
`my`	3
`SCHEME.IS`	3
`SCHEME_LIST`	9

SIZEOF	4
START	1
templateDescription	1
templateName	1
TypeConstraints	9
TypeConstraintsList	9
TypeFunction	9
TypeFunctionList	9
TypeName	9
VARIES_WITH	8

Lesson 1: Getting Started

This lesson describes how you can use the provided API to write TDL programs before we start describing the syntax of the language. We assume you have the AUTOMED API (http://www.doc.ic.ac.uk/automed/releases/index.html) already installed on you computer as well as the necessary resources required to run AUTOMED programs (see http://www.doc.ic.ac.uk/automed/ for details).

`TemplateCompilerSimulator` is an abstract class found in the AUTOMED API. Any TDL program should extend this class.



Template compiler simulator provides (amongst other things) a generic structure for a TDL program. The subclass should specify its details by defining certain abstract methods, which are summarized below:

Abstract method	**Purpose**
`String templateName()`	The name of the template that will be stored in the repository.
`String templateDescription()`	A description for the template, which will be stored in the repository
`String defaultModel()`	The name of a default model that the template is concerned with. Defining the model here gives a good syntactic shortcut when the program will be written.
`void defineTemplate()`	Here's where the TDL program is written

Here's an extremely simple program, which describes a template that doesn't do anything. It illustrates the minimum requirements that a program needs in order to work.

```
import uk.ac.ic.doc.automed.templates.wrapper.*;

public class EmptyTemplateProgram
       extends TemplateCompilerSimulator

{
      /* The constructor should have the following form*/
      public EmptyTemplateProgram()
            throws Exception
      {
            super();
      }


      /*Define the abstract methods*/

      protected String templateName()
      {
            return "Empty Template";
      }

      protected String templateDescription()
      {
            return "This is an test template that basically doesn’t do
                     anything";
      }

      protected String defaultModel()
      {
            return "er";
      }

      /*Define the template*/
      protected void defineTemplate()
            throws Exception
      {
            START();
            END();
            /* Don’t worry about what these mean */

      }

      public static void main(String [] args)
      {
            try
            {
                  new EmptyTemplateProgram();
            }
            catch(Exception e)
            {
                  e.printStackTrace();
            }
      }
}
```

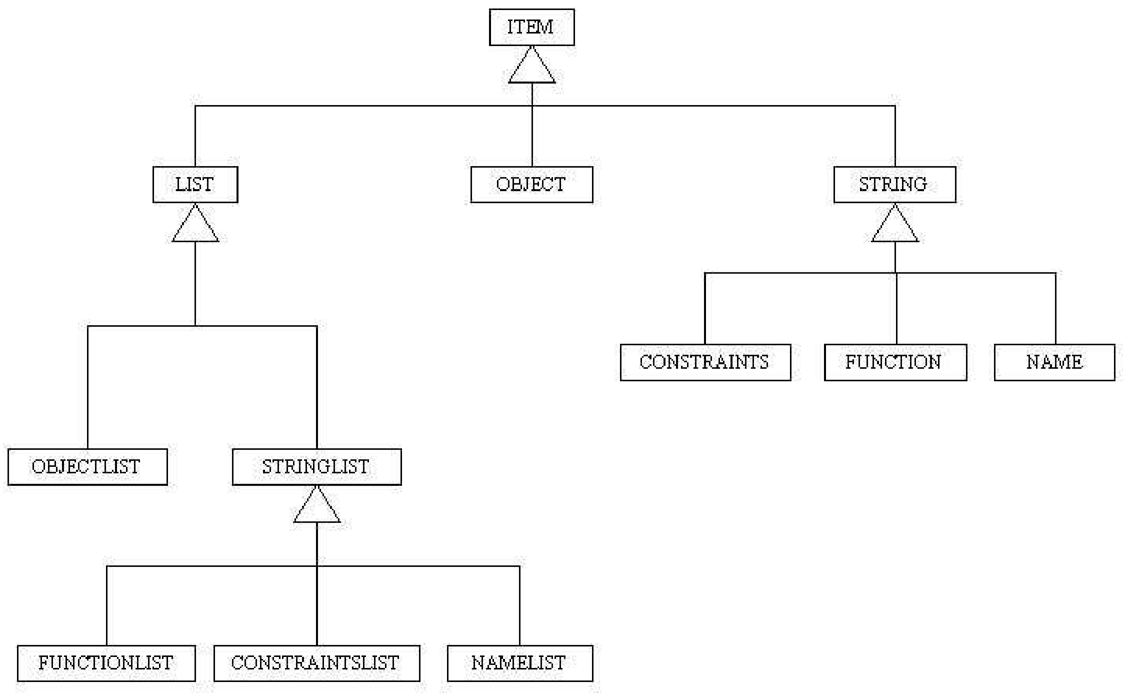
When does the program compile?

This question can be quite confusing since we are dealing with two different languages at the same time. We are writing a TDL program inside a java program. Since the TDL program is written in java, it should conform to the java constraints, but also to the TDL constraints. So compiling the program (using javac) will check whether the program is a correct java program. After doing that, running the java program will check whether it is a correct TDL program. So compiling it and running it, intuitively means that you are compiling it, which will in fact define the template in the repository. ***From now on when we say compile the TDL program we mean compile and run the java program.***

Lesson 2: Types

TDL is a typed language. There is a finite number of types which are static in the sense no user defined types are supported.

The following diagram illustrates the type hierarchy of TDL, and it will be a good reference for the lessons to follow.



Note that although you could play with casting (java allows that) the TDL program will not do what you think it would in such a case. ITEM is considered an ABSTRACT type in the sense that it cannot be instantiated in TDL.

Lesson 3: ADD statement

As you should know by now, the TDL program is written inside the `defineTemplate()` function. The idea is that the keywords of TDL are java functions.
Any TDL program should have the `START();` and `END();` keywords.

```
START();
END();
```

is a valid program that does nothing.

The statements of the program should be written inside the `START(); END();` keywords. This lesson explores the `ADD` statement, which adds an object in the schema. We will start describing this statement by writing a simple program.
Note that the **`EXTEND`** statement works exactly in the same way as the `ADD` statement but performs an extend instead of an add transformation.

It is customary for the first program demonstrated in any programming language to be the **Hello World** program and we are not planning to be an exception. TDL does not have the notion of output, so instead we shall write a program which adds an entity called Hello_World. We will write the entire class for the last time, after that we are only focusing on the `defineTemplate()` part. Here's how it goes:

```
import uk.ac.ic.doc.automed.templates.wrapper.*;


public class HelloWorld
      extends TemplateCompilerSimulator

{
      public HelloWorld()
            throws Exception
      {
            super();
      }

      //Define the abstract methods//

      protected String templateName()
      {
            return "Hello World";
      }

      protected String templateDescription()
      {
            return "Adds an entity called Hello_World";
```

```
	}

	protected String defaultModel()
	{
		return "er";
	}



	//Define the template//

	protected void defineTemplate()
		throws Exception
	{


		START();

			NAME entityName = DEFINE_NAME("Hello_World");

			OBJECT newEntity = ADD(CONSTRUCT.IS("entity"),
			                       SCHEME.IS(new Object []
			                                    {my(entityName)})
			                       );

		END();


	}

	public static void main(String [] args)
	{
		try
		{
			new AttributeEntityEquivalence();
		}
		catch(Exception e)
		{
			e.printStackTrace();
		}
	}
```

}

`DEFINE_NAME` creates a `NAME`. (For now just know that, as we will describe this statement later.) Here it creates the `NAME` with value "Hello_World";

An `ADD` statement takes minimum 2 arguments:

1. The **construct** of the `SchemaObject` that is added
2. the **scheme** of the object.

It returns an `OBJECT`: the object created after the statement is executed.

Now let’s take a look at the first two arguments, and how you could fill them in.

Construct

This defines the construct of the created object. In this example we said `CONSTRUCT.IS(“entity”);` This is because we have already said that our **default model** was the **“er”** model and in the repository “entity” is defined as a Construct of this model. This is actually one of the syntactic shortcuts of the language.
An alternative way of giving the construct is to say (at the beginning of the function):

```
Model er = Model.getModel(“er”);
Construct entity = er.getConstruct(“entity”);
```

and then, instead of writing `CONSTRUCT.IS(“entity”),` you just write `entity`, or `CONSTRUCT.IS(entity);`

Note that when you `CONSTRUCT.IS` is a polymorphic function. When you give it a `String` as an argument it tries to return the `Construct` given as `defaultModel.getConstruct(“string”);` When you give it a `Construct` as an argument it just echos the argument. The only reason you want to use `CONSTRUCT.IS(entity)` instead of just `entity` is for readability.

Scheme

This defines the scheme of the object that is added (for more on how scheme is used in automed consult the automed website and the API documentation).

The scheme is defined as an array of `Objects` (java Objects). The construct of the `OBJECT` that is added dictates the structure of the scheme. Each member of the array can be of the following forms:

a) An `OBJECT`: When an other `SchemaObject` is referred to at the particular position of the scheme
b) A `STRING` (a TDL `STRING` not a java `String`): When at this position in the scheme is `String` type (for example: *“1:1”*, *“total”* etc)
c) A java `String`: Used as above.
d) A `LIST`: When this position of the scheme has an unbounded upper limit. For example when this position holds the subentitites of the generalization. Since the number of subentities varies, you create an `OBJECTLIST` that holds all subentities, and use this list at the particular position.
e) **`my(NAME):`** This is a special case of where you use a name. When you use a `NAME` that is the name of the object that is added you need to point it explicitly by saying `my(entityName)` for example. If you just say `entityName` you will get an invalid scheme exception. All schemes should contain one of these.

In the example above the object added was an `entity`. Entity scheme only holds the name of the entity. So we could define the scheme as:

```
new Object[]{my(entityName)}
```

which is completely valid. However for readability purposes you might want to say: `SCHEME.IS(new Object[]{my(entityName)})`

`SCHEME` and `CONSTRUCT` are mandatory parameters and should be used in whenever ADD is used. Besides these however, you could define a `FUNCTION` and `CONSTRAINTS` of the transformation. You could use a function by:

a) Using a `FUNCTION` directly
b) Using a `FUNCTION` f and saying `FUNCTION.IS(f)`
c) Using a java String and saying `FUNCTION.IS(“function definition .. ”);`

the same goes for `CONSTRAINTS`.

The overloading of the ADD function is shown in the following table (extracted from the API documentation)

protected OBJECT	**ADD**(Construct con, Object[] scheme)
protected OBJECT	**ADD**(Construct con, Object[] scheme, CONSTRAINTS c)
protected OBJECT	**ADD**(Construct con, Object[] scheme, FUNCTION f)
protected OBJECT	**ADD**(Construct con, Object[] scheme, FUNCTION f, CONSTRAINTS c)
protected OBJECT	**ADD**(Construct con, Object[] scheme, CONSTRAINTS c, FUNCTION f)

Here are some examples of how add is used:

```
OBJECT newEntity = ADD(entity,
                   SCHEME.IS(new Object [] {my(subName)}),
                   FUNCTION.IS(f1)
                  );


ADD(CONSTRUCT.IS(“entity”),
                   SCHEME.IS(new Object [] {my(subName)}),
                   FUNCTION.IS(f1),
                   CONSTRAINTS.IS(c1),
                  );
```

```
ADD (CONSTRUCT.IS(“generalisation”,
      new Object [] {my(genName),
                  "total",
                  parentEntity,
                  createdEntities
               }
);

OBJECT newRel = ADD(CONSTRUCT.IS("relationship"),
                    SCHEME.IS(new Object[]{my(newRelationshipName),
                                           existingEntity,
                                           newEntity,
                                           "1:1",
                                           "1:N"})
                      );
```

An `OBJECT` that is added enters the **current scope** and can be used by other `OBJECT`s later.

When an ADD transformation is implemented, the new object is added to a **new Schema**. This is true for all tranformations: they take you from a source schema to a target schema. For the lower layers of the template definition API, linking from one schema to the next should bee done explicitly. Here this is done on the background and hidden from the user.

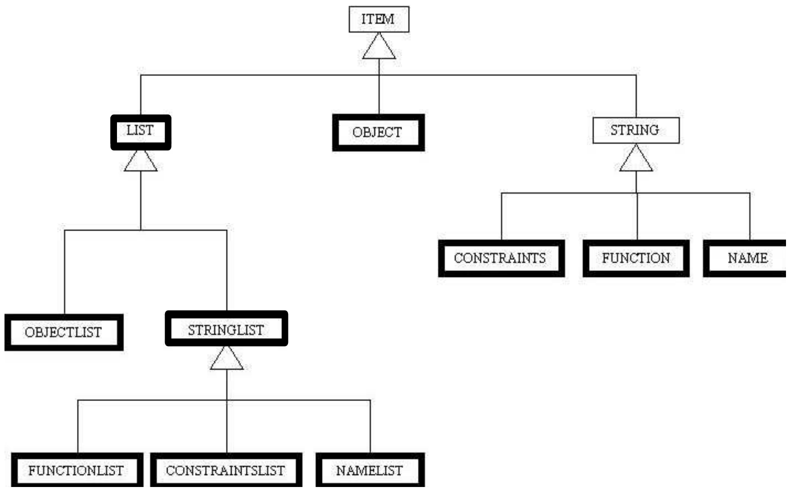
Lesson 4: Template Inputs

We have already explained that the statements of a TDL program are inside the `START();/END();` keywords. Optionally, before `START();` You could define the **inputs** of the template. Inputs are defined after the `INPUTS();` keyword. So the structure of the program is:

```
INPUTS();
 // define the inputs here
START();
 // the program
END();
```

The **default input** for each template is the **initial schema**. This is hidden from the programmer, but when the template is executed the first input that it asks for is the initial schema id. However the programmer can ask for more inputs.

Each input has a type and you can have arguments of any of the **<u>leaf</u>** types. The following diagram highlights the types that an input can have:



You define inputs using `askFor` statements. There is a different statement for each type. for example:

```
askForObject
askForConstraints
askForObjectList
askForConstraintsList
```
etc

The first argument of `askFor` statements is always the String that will prompt the user to insert the particular input argument. For example:

```
askForName("Please insert the name of the entity that will be added");
```

Before we start describing these statements, here's the definitions of the askFor functions taken from the API documentation:

protected CONSTRAINTS	**askForConstraints**(String description)
protected FUNCTION	**askForFunction**(String description)
protected NAME	**askForName**(String description)
protected OBJECT	**askForObject**(String description)
protected OBJECT	**askForObject**(String description, Construct constr)
protected CONSTRAINTSLIST	**askForConstraintsList**(String description)
protected FUNCTIONLIST	**askForFunctionList**(String description)
protected NAMELIST	**askForNameList**(String description)
protected OBJECTLIST	**askForObjectList**(String description)
protected OBJECTLIST	**askForObjectList**(String description, Construct constr)
protected CONSTRAINTSLIST	**askForConstraintsList**(String description, LIST ref)
protected FUNCTIONLIST	**askForFunctionList**(String description, LIST ref)
protected NAMELIST	**askForNameList**(String description, LIST ref)
protected OBJECTLIST	**askForObjectList**(String description, LIST ref)
protected OBJECTLIST	**askForObjectList**(String description, Construct constr, LIST ref)

Ask for Constraints, Function, Name

This is the simplest case. The *description* is the only required argument. For example:

```
NAME entityName = askForName(“Insert name of entity to be added”);
FUNCTION func = askForFunction(“Insert the function for the
                                                transformation”);
CONSTRAINTS cons = askForConstraints(“Insert the constraints of the
                                                transformation”);
```

Note that the created `ITEMS` can be used later in the program.

Ask for Object

```
OBJECT obj = askForObject(“Insert an object”);
```

The will ask for an object of **any construct**. If you wish to specify a construct you will have to use a second parameter.

```
OBJECT obj = askForObject(“Insert an entity”, CONSTRUCT.IS(“entity”));
```

(The way Constructs are used is explained in previous lesson (ADD statement))

Ask for Lists

In a similar way to how you ask for single ITEMs, you can ask for lists of ITEMs. For example:

```
NAMELIST subentities = askForNameList(“Insert the names of
                                                subentities”);
OBJECTLIST attributes = askForObjectList(“Insert some attributes”,
                                          CONSTRUCT.IS(“attribute”));
```

Sometimes several lists should be of the **same size**. For example you might want a list of entities and a list of names, one for each entity. So you would want the two lists to be of the same size. One way to do it is to trust the common sense of the user and pray that he will understand from the description that the two lists should be of the same size. On the other hand you might want to enforce this constraint at compile time. In this case you should add an extra argument at the end of the second ask for statement to say that it should be of the same size as the first. For example:

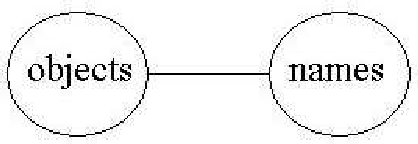
```
OBJECTLIST objects = askForObjects(“Insert some objects”);
```

```
NAMELIST names = askForNames(“Give the names of the objects”, objects);
```

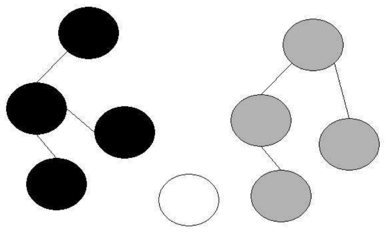
the second statement says ask for a list of the same size as the objects list.
For readability you could instead say:

```
NAMELIST names = askForNames(“Give the names of the objects”,
                              SIZEOF(objects));
```

We could have **more than two lists** having the same size. To understand what happens in this case, think of the lists as nodes, and think of referencing one list from an other as adding an arc from one node to the next. For example the above created the following graph.



The rule is: A list has the same size as all the lists that are reachable from it. For example in the following diagram, the lists with the same color have the same size.



Lesson 5: DELETE

You can delete objects that are in the current scope. **After you delete the object it is removed from the current scope**. So for example you cannot delete it and then rename it or use it in the scheme of an other object. Note that the **CONTRACT** statement works exactly like the DELETE statement

The following program asks for an object and then deletes it.

```
INPUTS();
OBJECT obj  = askForObject("Object to be deleted");
START();
DELETE(obj);
END();
```

If you try to use obj after the DELETE statement, it will complain for trying to access an ITEM that is not in the current scope.

A more readable way of saying DELETE(obj); is:
obj.DELETE();
Which does the same thing.

You could add two more arguments to the DELETE statement, namely the FUNCTION and CONSTRAINTS, used in a similar way as the ADD statement (see lesson 3 for more).

Examples:

```
obj.DELETE(FUNCTION.IS(f1), CONTRAINTS.IS(c1));
DELETE(obj, FUNCTION.IS(f1), CONSTRAINTS.IS(c1));
obj.DELETE(CONSTRAINTS.IS(c1));
```

The overloading of the DELETE function is shown below:

protected void	**DELETE**(OBJECT obj)
protected void	**DELETE**(OBJECT obj, CONSTRAINTS c)
protected void	**DELETE**(OBJECT obj, CONSTRAINTS c, FUNCTION f)
protected void	**DELETE**(OBJECT obj, FUNCTION f)
protected void	**DELETE**(OBJECT obj, FUNCTION f, CONSTRAINTS c)

Lesson 6: RENAME

This statement to rename `SchemaObjects` and it is quite straight forward. It takes the `OBJECT` to be renamed and the **new name** as arguments. The following example renames an object with a name given as a paremeter.

```
INPUTS();
OBJECT obj = askForObject(“Object to be renamed”);
NAME newName = askForName(“New name given”);

START();
RENAME(obj, newName);
END();
```

Similar to `DELETE`, you could rename the object by saying:

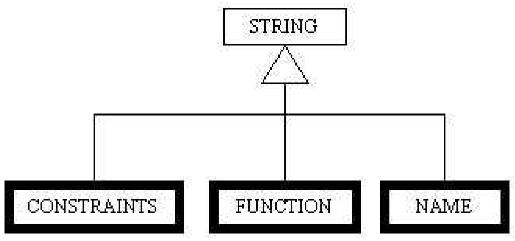
```
obj.RENAME(newName);
```

If you knew the new name at compile time you could use a java String instead of a `NAME`. For example:

```
RENAME(obj, “john”);
obj.RENAME(“john”);
```

Lesson 7: String manipulation in TDL

Strings and string manipulation in TDL is often confusing partly because there is often the issue of what is a java `String` and what is a TDL `STRING`. But before I frighten you even more, recall on which items are considered a TDL `STRINGs` by looking at the type tree below.



There are 4 main statements that have similar structure, one for each type:

```
DEFINE_STRING
DEFINE_CONSTRAINTS
DEFINE_FUNCTION
DEFINE_NAME
```

The difference between them is merely to the type of `STRING` they produce, from then on if you know one of them you know all of them, that's why we shall not comment on each one of these independently but instead use them interchangeably.

There are two ways to use these functions: Using `Object` arrays, and using `Strings`. They both have the same expressive power, however the second one is much more elegant. We will however explain the first way first as it is simpler and we will later explain how the second way is derived form the first.

<u>Define STRINGS Using Arrays</u>

The main idea here is that you create a `STRING` by concatenating other `STRINGs`. For example, let's say that we somehow managed to define two Strings:

```
STRING hello; //has the value “hello_”
STRING world; //has the value “world”
```

We can concatenate these two into a new string by saying:

```
NAME hw = DEFINE_NAME (new Object [] {hello, world};//”hello_world”
```

The `Object[]` could have any STRING as its elements. Furthermore it could also have any `java String`. So for example:

```
FUNCTION hw = DEFINE_FUNCTION(new Object [] {“hello_”,
                                             world,
                                           “ how are you”,
                                           ”?” } );
```

hw now has the value “hello_world how are you?

Note that the `STRING world` could have come from an `Input` so its value can be dynamic.

Define String Using Java String

If we want to define string `hello_world` we could just say:

```
STRING hw = “hello_world”;
```

But we would like to do the concatenation stuff we did in the previous approach. Saying:

```
STRING hw = “hello_”+world+“how are you”+“?”  ;
```

is not appropriate, as it will just take the string representation `(toString())` of the `world` variable and form a new static `String` (remember that the TDL program compiles when the java program runs).

Now consider the following program. What it does, is asks the user for her name and define a `STRING` that could for instance say: “User’s name is Julie”

```
INPUTS();
      STRING name = askForName(“What is your name?”);
START();
      ALIAS(name, “name”);
      STRING sentence = DEFINE_STRING(“User’s name is @name”);
END();
```

What **`ALIAS`** does, is assign an alias java `String` to an `ITEM`. For example here the `String “name”` is an alias for the `STRING name`. Here we decided to call it `“name”` but we could call it anything of the correct format:

Aliases format:
- Made up only of alphanumeric characters
- Have a length of at least 1.

An `ITEM` can have any number of aliases, but an alias can only be assigned to a single `ITEM`. You can only define aliases for `ITEMS` that are in the current scope.

```
STRING sentence = DEFINE_STRING(“User’s name is @name”);
```

is equivalent to:

```
STRING sentence = DEFINE_STRING(new Object[] {“User’s name is”, name});
```
and in fact when the string above is parsed this is what it returns.

If you want to clearly define the name of the alias, you can include it in brackets, e.g.

```
STRING sentence = DEFINE_STRING(“User’s name is @(name)”);
```
is equivalent to the previous example. This is useful when the rest of the sentence after the alias contains alphanumerical characters, e.g.

```
NAME primarykeyName = DEFINE_NAME("@(specializationTableName)_pk");
```

Having defined the alias you can use it in a `String` such that `@alias` refers to the item that the alias is aliasing. For now, just assume that you can only use aliases of `OBJECTs` and `STRINGs` in the `String`.

Using an `OBJECT` in the String will actually place the ID of the `SchemaObject` in the repository. Eg.

```
STRING s = DEFINE_STRING(“entity is @myEntity”);
```

will return something like: `“entity is 142”;`

I can’t think of any real application where this could be of any use. However there are a coulple of tricks we can do that makes the use of object reallyuseful:
`?name` extention and
`?scheme` extension

For example:

```
STRING s1 =DEFINE_STRING(“attribute name is @myAtt?name”);
STRING s2 = DEFINE_STRING(“attribute scheme is   @myAtt?scheme”);
```

s1 will give something like `“attribute name is p_code”`
s2 will give something like `“attribute scheme is <<person, p_code>>”`, depending of how the scheme of the attribute is defined in the repository.

Concatenating Stringlists

An other `ITEM` instance we can use in a `DEFINE_STRING` statement is a `STRINGLIST`. What it does is concatenate all the elements of the STRINGLIST. For example:

```
INPUTS();
      STRINGLIST fruits = askForStringList(“Which fruits do you
                                                       like?”);
START();
      ALIAS(fruits, “fruits”);
      STRING f = DEFINE_STRING(“User likes: @fruits”);
END();
```

If the user entered the list:

Apples
Pears
Figs

`f` would have the value: `“User likes: ApplesPearsFigs”`

I am sure you can think of numerous tasks where the above example can be applied to a real Database Schema Integration process (ok, stop laughing), however when we explain how looping works you will really realize the use of this feature.

There is also the option to define the substring that will appear in between the strings that are concatenated. All the DEFINE_STRING methods are overwritten with an extra artument (String) which defines the concatenation string to be used. For example, the following STRING f

```
STRING f = DEFINE_STRING(fruits, “++”);
```

would have the value `“Apples++Pears++Figs”`. This can be very useful when concatenating IQL queries.

Lesson 8: Looping in TDL, the FOREACH statement

Looping in TDL is done using a **`FOREACH`** statement. A `FOREACH` iterates through a list and can use the current element of the list at each iteration.
The general structure of the statement is the following:

```
FOREACH();
     //Foreach Header
     DO();
     //Loop Block
ENDFOREACH();
```

VERY IMPORTANT

The **scope** of the Loop Block contains **ONLY** the variables declared at the **Foreach Header**. If you want to use a variable that was on the scope before the loop, you have to explicitly re-declare it (using **`VARIES_WITH`** see later) on the Foreach Header. Variables declared inside the loop body are not on the scope outside the loop (except `COLLECT` lists see later).

The following program asks the user for a list of names, and adds entities with those names, one for each name

```
INPUTS();
      NAMELIST entityNames = askForNameList("Names of entities to
                                                            add");

START();

      FOREACH(); NAME currentName =IN (entityNames);
            DO();
                  OBJECT newEntity = ADD(CONSTRUCT.IS(entity),
                                     SCHEME.IS(new Object []
                                                {my(currentName)}),
                                    );
      ENDFOREACH();

END();
```

The main header statement is the **`=IN`** statement.
Seeing `IN` as a java method, it takes a `LIST` and returns an `ITEM` of the same type as the elements of the list. Here’s the overloading of the function

protected CONSTRAINTS	IN(CONSTRAINTSLIST conList)
protected FUNCTION	IN(FUNCTIONLIST funList)
protected NAME	IN(NAMELIST nmList)
protected OBJECT	IN(OBJECTLIST obList)

In the example above, it takes a `NAMELIST` and returns a `NAME`.

```
NAME currentName =IN (entityNames);
```

this says that `currentName` iterates through the list `entityNames`, and takes the current value of the list at each iteration of the loop. The loop is iterated as many times as the number of elements in the list `entityNames`.

You can iterate through more than one list at any time. You could say:

```
FOREACH(); NAME currentName =IN(entityNames)
           OBJECT currentAttr =IN(attributes)
```

The lists `entityNames` and `attributes` should have the same size, otherwise the program might misbehave. If these lists come from the inputs, you could enforce this by using the `SIZEOF` option (see lesson for inputs).

VARIES_WITH

As we said above, if you want to use an item that comes from outside the loop inside the loop, you will have to **re-declare** it on the **loop header**. This is done using the **`VARIES_WITH`** statement.

For example:

```
OBJECT parentEntity = …

FOREACH(); NAME currentName =IN(subentities);
            OBJECT parent =VARIES_WITH(parentEntity);
            DO();
            …
ENDFOREACH();
```

So `parent` is basically the same object as `parentEntity` as far as the programmer is concerned.

Collecting Inside the loop

We have already mentioned that what is created inside the loop is not accessible outside the loop. This would mean that you cannot use things that are created by the loop outside of it.

This is true. However you can use the **COLLECTS** statement, to create lists that collect objects that are produced inside the loop.

```
FOREACH(); NAME currentName =IN (entityNames);
      DO();
      OBJECT newEntity = ADD(CONSTRUCT.IS(entity),
                             SCHEME.IS(new Object []
                                          {my(currentName)}),
                              );
      OBJECTLIST createdEnts =COLLECTS(newEntity);
ENDFOREACH();
```

Each time the loop is iterated an entity is created, and added to the `createdEnts OBJECTLIST.` Collecting lists are accessible outside the loop. `COLLECTS` statements should be the **last statements** of the loop block but you can have **more than one** `COLLECTS` statements.

Nested Loops

Nested loops are supported in TDL. There is nothing special with nested loops. Just remember that if you want to use a variable that is outside the loop in the internal loop, you will have to re-declare it on the header of the outer loop and re-re-declare it on the header of the internal one.

From what we learned up to now you should be able to understand and reproduce the following TDL code that implements an Attribute to Generalisation equivalence template transformation (not complete as it needs some more functions).

```
INPUTS();
      OBJECT parentEntity = askForObject("Existing parent entity",
                                          entity);
      OBJECT existingAttribute = askForObject("Attribute to be
                                                      deleted",
                                                attribute);

      NAMELIST subEntityNames = askForNameList("Names of the
                                                      Subentities");
      NAMELIST populationOfEntities = askForNameList("Values of the
                                          attribute that correspond to
                                          subentities",
                                          SIZEOF(subEntityNames));
      NAME genName = askForName("Generalisation Name");


START();

      FOREACH(); NAME subName =IN (subEntityNames);
               NAME popName =IN (populationOfEntities);
               OBJECT att =VARIES_WITH(existingAttribute);
               OBJECT parent =VARIES_WITH(parentEntity);

            DO();
                  ALIAS (popName, "nameGiven");
                  ALIAS (subName, "subname");
                  ALIAS (att, "att");
                  ALIAS (parent, "parent");

                  FUNCTION f1 =
                        DEFINE_FUNCTION("@subname(X):-
                                           @att?scheme(X,@nameGiven)");

                  OBJECT newEntity = ADD(CONSTRUCT.IS(entity),
                                     SCHEME.IS(new Object []
                                                      {my(subName)}),
                                     FUNCTION.IS(f1)
                                    );
                  OBJECTLIST createdEntities =COLLECTS(newEntity);

      ENDFOREACH();



      ADD (generalisation,
           new Object [] {my(genName),
                        "total",
                        parentEntity,
                        createdEntities
                     }
      );



      existingAttribute.DELETE();
END();
```

Lesson 9: Manipulating the Scheme of Objects

Up to this point, TDL is (relatively) strongly typed. The programmer is not allowed to play with types and therefore we are guaranteed (to some extend) that once the TDL program compiles then the program will run well (remember that TDL program compiles means the java program runs).

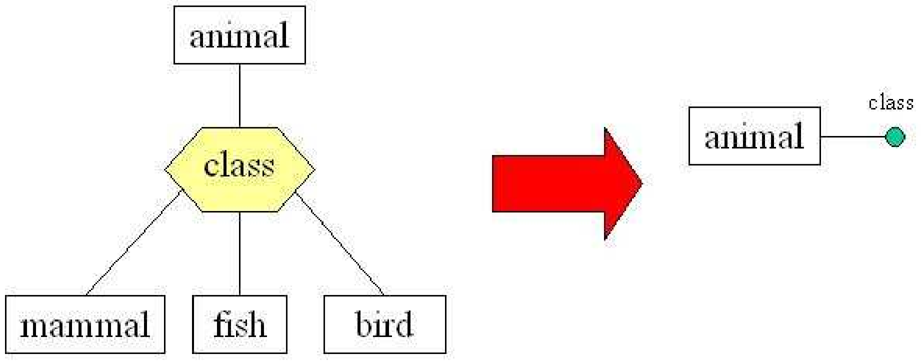
This lesson introduces the statements:

- `SCHEME_LIST :` Creates a list containing the scheme items of an object.
- `ELEMENT_AT :` Accesses an item of the list at a particular position. The type of the object has to be explicitly stated when the statement is used.
- `CREATE_LIST :` Creates an arbitrary list of `ITEM` objects. Its main use is the definition of an object's scheme.

Someone should use these statements only if she is sure how the scheme of each object is defined inside the repository.

I could spend quite some time talking about these statements but I think I would save quite some ink and paper if I described what is going on using an example.

Assume we would like to write a template that resolves Generalization Attribute equivalence by replacing the generalization with an attribute with the same name as the generalization. An instance of this transformation is illustrated below.



In order to perform this transformation we will have to know:

- The parent entity (animal)
- The generalization (class)
- The subentities (mammal, fish, bird)

With the statements we have explained up to now, in order to perform this transformation we will have to ask the user to enter the parent entity, the generalization and a list with the subentities.

However, the parent entity and the subentities, are all part of the **scheme** of the generalization. So if we could access the scheme of objects, then in fact all we would only require from the user is to give is the generalization.

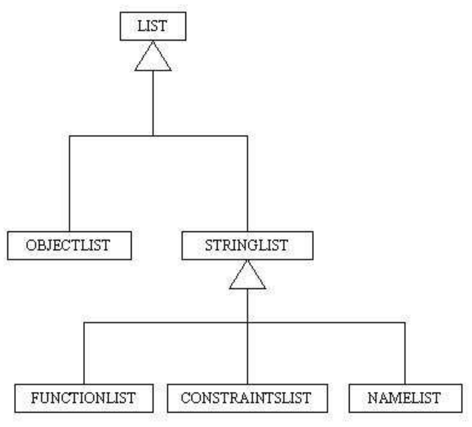
Here's an example of the scheme of an `SchemaObject` (generalization) as defined in the Automed repository (for more on schemes see the Automed website).

Index	Scheme Object at this position	Type	Example
0	name of generalization	string/single	"class"
1	type of generalization	string/single	"total"
2	parent entity	object id/single	animal:ID
3	subentities	object ids/ list	list: (mammal:ID, fish:ID, bird:ID)

This can be seen as a list with a different element at each position. As you can see different positions have different types. Abstracting the types in the table above to TDL types, then indices 0 to 4 would have the following types:

Index	TDL type
0	STRING
1	STRING
2	OBJECT
3	OBJECTLIST

I will tell you now that you can obtain the Scheme List using the `SCHEME_LIST` statement. The main problem now is what type should this list be? I remind you that the list types are the following:



None of the leaf types can suit the type of the scheme list. The scheme list is therefore of type `LIST` and we give freedom to the programmer to specify what type each element has.

To make things even worse, The `LIST` given by the `SCHEME_LIST` contains **lists** of one element instead of single items. The table below summarizes what you will really get when you get the scheme list:

Type at scheme	**What you get**
OBJECT	An OBJECTLIST of one element that contains the object at index zero
STRING	A STRINGLIST of one element that contains the string at index zero
OBJECTLIST	OBJECTLIST

In other words, in order to get the parent entity, you should get the `OBJECTLIST` at position 2, and from that take the `OBJECT` at position 0.

Getting the scheme list can be done quite simply:

```
LIST schemeList = SCHEME_LIST(myGeneralisation);
```

Or

```
LIST schemeList = myGeneralisation.SCHEME_LIST();
```

Note that you cannot use the `schemeList created` in the `foreach` statement as you are only allowed to iterate a LIST but only the leaf types of LIST.

The only way you can use it is using the **`ELEMENT_AT`** statement. Since we don't know what the element at each position is, we trust that the programmer will explicitly say what the type of the element is and that this type will be correct.

So, `ELEMENT_AT` is used in the following way:

```
CONSTRAINTS c = (CONSTRAINTS) aList.ELEMENT_AT(0,TypeConstraints);
FUNCTION c = (FUCNTION)aList. ELEMENT_AT(3,TypeFunction);
OBJECT c = (OBJECT)aList. ELEMENT_AT(5,TypeObject);
STRING c = (STRING)aList. ELEMENT_AT(2,TypeString);
STRINGLIST c = (STRINGLIST)aList. ELEMENT_AT(2,TypeStringList);
```

And so on, for all types (except `ITEM` and `LIST`). Note the how **casting** is used, and how the elements are **indexed from position zero**.

When the **Construct** of the `OBJECT` in the list, it is better to specify it by:

```
OBJECT entity = (OBJECT)aList. ELEMENT_AT(5,CONSTRUCT.IS(“entity”));
```

So far, we’ve seen how to get `LIST`s from existing methods, e.g. `SCHEME_LIST()`. In order to create your own `LIST`s, you need to use the `CREATE_LIST` statement. It has a single argument which is an array of `ITEM`s that constitute the list.

This statement is useful for defining the scheme of an object and especially when you’ve got nested lists. For example, to create the scheme of a primary key construct you need a list of all the columns of the table that are part of the primary key:

```
OBJECTLIST primaryKeyColumns =
             (OBJECTLIST)CREATE_LIST(new ITEM[]{newColumn});
ADD( CONSTRUCT.IS(primarykey),
          SCHEME.IS(new Object[]{my(primarykeyName),newSpecialization,
primaryKeyColumns})
    );
```

Now take a look at the **Generalization to Attribute Equivalence** we promised you. All the syntax has been described already. Note how we first take the `OBJECTLIST` that contains the parent entity and use that to take the first element, ie the parent entity itself

```
INPUTS();
      OBJECT generalisation = askForObject("Existing parent entity",
CONSTRUCT.IS("generalisation"));

START();
      LIST genScheme = generalisation.SCHEME_LIST();
      OBJECTLIST parentList = (OBJECTLIST)genScheme.ELEMENT_AT(2,
TypeObjectList);
      OBJECT parent = parentList.ELEMENT_AT(0, CONSTRUCT.IS("entity"));

      OBJECTLIST subEntities = (OBJECTLIST)genScheme.ELEMENT_AT(3,
TypeObjectList);

      generalisation.DELETE();

      FOREACH(); OBJECT subEntity =IN(subEntities);

            DO();
                  subEntity.DELETE();

      ENDFOREACH();

      ADD(CONSTRUCT.IS("attribute"),
              SCHEME.IS(new Object [] {parent, my(attName), "key"})
       );
END();
```

3.3.2 Runtime Template Transformations

The discussion in the previous section was focused on *static* template transformations. They are static because a special Java program needs to be written for each one of them, with specific methods that should override methods of the `TemplateCompilerSimulator`. (see Lesson 1).

The AutoMed API enables also the definition of *Runtime Template Transformations*. There is a class `DefaultTemplate` which can be used for this purpose. To create a template transformation at run-time, you need an instance of the `DefaultTemplate`, which you set to have the arguments and statements you need. The syntax is exactly the same as described in the previous section.

A sample of the code that creates a run-time template transformation follows:

```
DefaultTemplate t = new DefaultTemplate(er, “runtime template”, “”);

t.INPUTS();
  TemplateCompilerSimulator.NAME entityName =
     t.askForName("New entity name");

t.START();
  TemplateCompilerSimulator.OBJECT newEntity=
     t.ADD(entity, new Object[]{t.my(entityName)});
t.END();
```
