The Spider Environment

Leonardo Mendonça de Moura
e-mail: moura@les.inf.puc-rio.br

Carlos José Pereira de Lucena
lucena@inf.puc-rio.br

Arndt von Staa
arndt@inf.puc-rio.br

PUC-RioInf.MCC46/97 December, 1997

Abstract: Visual composition is an interactive development of different applications by the direct manipulation of reusable components. We believe that the visual composition approach deals directly with the complexity of large software systems making their development easier, more flexible, and easier to be understood. This is accomplished by implementing abstraction, reuse, and visualization concepts. The developer becomes a component builder and no longer creates large applications that are hard to maintain and enhance. The user will have the freedom to choose the components he needs to build an application, and to mix and match components from different developers until the desired functionality is achieved. The visual Spider environment allows the creation of different kinds of applications.

Keywords: visual composition, components, domain oriented software development, black-box reuse

Resumo: Composição visual pode ser definido como o desenvolvimento interativo de diferentes aplicações pela manipulação direta de componentes reutilizáveis. Nós acreditamos que a abordagem composicional lida diretamente com a complexidade de grandes sistemas de software tornando o desenvolvimento mais fácil, reusável, e de fácil compreensão. Isto é alcançado pelo uso de abstrações, visualizações e reuso. O desenvolvedor torna-se um criador de componentes e não mais cria grandes aplicações que são difíceis de serem mantidas e aperfeiçoadas. O usuário tem a liberdade de escolher os componentes necessários para construir um aplicação.

Palavras-chave: composição visual, componentes, desenvolvimento orientado a domínio, reuso de caixa-preta

1. Introduction

The Spider environment [17, 19] has changed the textual paradigm of programming into a visual paradigm [5] of direct manipulation and edition of application and user interface components. Spider stimulates a component oriented approach [20] of software development where collections of objects can be reused and built in an evolutionary software life cycle. The environment's main objective is to give support to the construction of large applications starting from prefabricated components in a given application domain.

Visual programming provides an environment where programs — not only user interfaces — are built by the direct manipulation of icons. Visual composition joins the visual paradigm with a component oriented approach. We can also define visual composition as the interactive development of applications by the direct manipulation of reusable components. This approach deals directly with the complexity of large systems. The complexity is controlled in such a way that the system becomes easier to be developed, tested, and debugged, increasing flexibility and understanding. That is achieved by the use of abstractions specially tailored for a given domain, reuse and visualizations. The objective is to have the developer as a creator of components and not as the creator of large applications that are difficult to maintain and to develop.

The applications are built by the composition of components created by different developers that together define the functionality they want. In visual composition the programmer's role is minimized. The focus is on software composition and reuse. Conventional programming languages are used to develop reusable software components and not to develop specific applications. What one calls "components" are the main computational elements of the Spider environment. Components are high level objects built through textual or visual programming. Components defined by text (primitive components) are implemented in conventional and/or script languages. Components defined by pictorial elements (composite components) are implemented by the direct manipulation of other components. The components define what we may call a "construction block library", from which one can choose the elements one needs from a wide variety of components.

As described above, the software development process in the Spider environment has two main roles. The first is related to the development of reusable components, this role is performed by a special developer that we will call "fabricator". The other is related to the construction of application by visual composition. This role is performed by the developer who we will call "builder". We believe that fabricators should be developers who are highly specialized, and builders may be less so. The visual paradigm works as a bridge between fabricators and builders.

We observe that current textual and visual languages are not expressive enough to represent the richness in semantic properties and development features of different application domains. In order to cope with this problem, we have been introduced the "architecture" concept in the environment. Architecture is an implementation model for a given application domain, for which the abstractions, composition rules and editor behavior have been specially tailored.

Fabricators build architectures based on domain analysis [2,6]. Using domain analysis as a starting point; fabricators build and select appropriate components. When a builder uses an architecture, he selects the components that will help him to build the application. The component composition is restricted to the composition rules defined by the fabricators for the application domain. Architectures are composed of: a set of components, composition rules, layout rules, wizards and tools. The wizards are created to help the builders in the application construction, and the tools are a kind of macro that are used to automate repetitive tasks.

Whereas on the one hand fabricators are seen as the builders of different environments for particular application domains, on the other builders are seen as the users of these environments.

We have structured the other parts of this paper in the following manner: in section 2, we will describe the Spider environment elements; section 3 will deal with environment implementation details; in section 4, we will describe application samples which have been built by using the environment; details on how to integrate different architectures will be analyzed in section 5; in section 6, we will discuss other projects on the subject we intend to develop; in section 7, we will discuss what we are working on and what intend to do in the near future. The conclusions will be shown in section 8.

2. Spider Environment Elements

The main module of the Spider environment is the composition editor, that is used by the builder to create applications. We can see this editor as a graph manipulator, specialized in visual composition. As mentioned before, the Spider environment contains the architecture concept. Thus the behavior of the composition editor and the components that are available will vary depending on the specific application domain. The fabricator must customize the editor for a given architecture. The components are the basic elements manipulated by the composition editor. These are black boxes with a well defined interface (protocol). One of the most important parts of a component is its connection ports. The ports define what kind of connections may be made between components. Components also have properties that are used to set up its initial state at run-time.

The environment allows the fabricators to define the architectures so that the builder may create components by using the composition editor. The components created by the use of the composition editor are called composite components.

All the architectures that have so far been built allow the creation of composite components. We feel that that is a powerful mechanism available to builders so that they may be able to improve application modularity. Composite components are built from other components (internal components). A composite component defines the connections between the set of internal components. A connection between the ports of two components defines a kind of relationship between them (Figure 1).

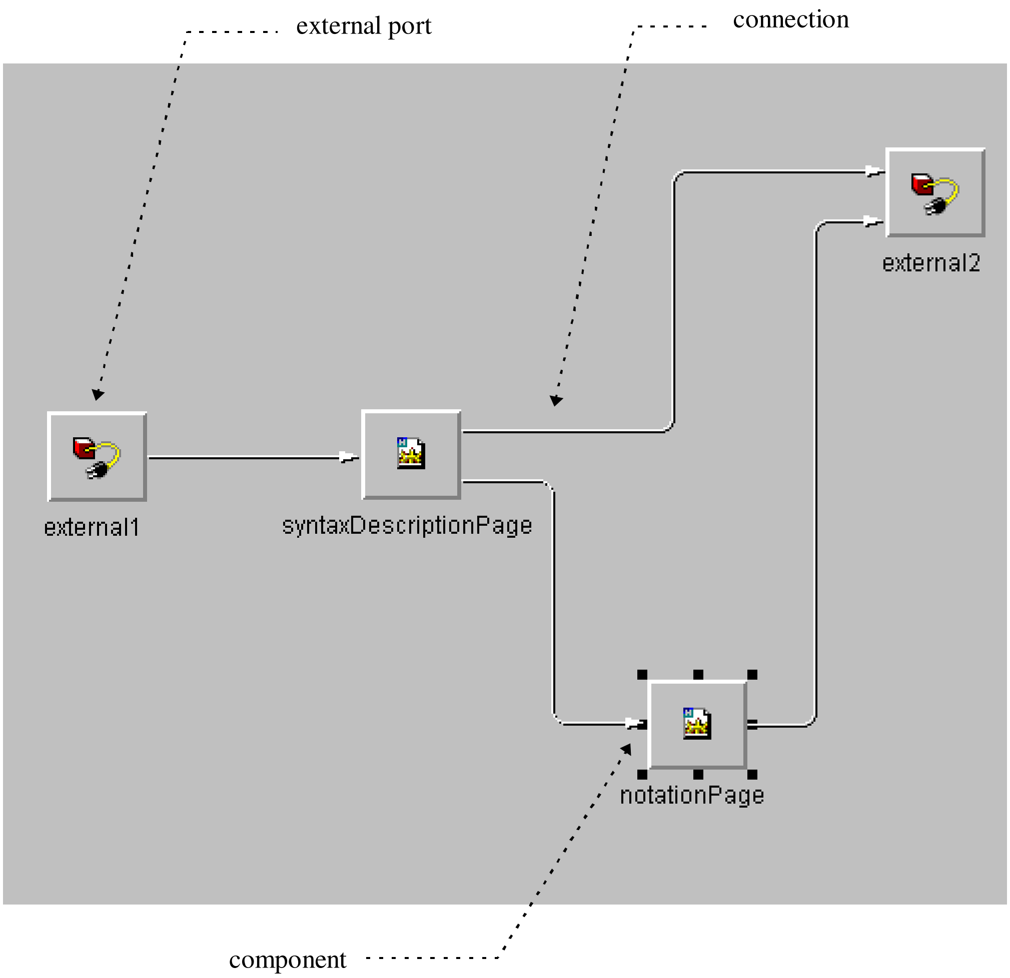

Figure 1 Composite Component

Internal components are either primitive or composite. Primitive components may be implemented in conventional programming languages such as Java or C++. The technique used to implement primitive components is chosen by the fabricator when defining the architecture. Therefore, the fabricator will implement primitive components depending on the architecture he has defined. If a composite component is used in the construction of another component, only an external vision (black box) is provided.

A composite component encapsulates a group of components that have been partially connected and determines which ports will be visible for other components. We note that the composite component ports function as bridges to the internal component ports. The composite component input ports should be connected to the internal component input ports. The same is valid for output ports.

Composite components are created to encapsulate behaviors and private services. They are reusable, time-saving designs; so that the developer doesn't need to reconstruct the same pattern several times. We can also see composite components as examples of how to construct a component with some functionality, so that they may be modified by the addition, deletion or modification of internal components.

Bertrand Meyer defines the weak coupling or small interfaces as follows [14]: "If any of two modules communicate at all, they should exchange as little information as possible." He justifies this principle in particular by the criterion of continuity and protection, which demands that changing a small part of a module's interface should result in little modification of other parts of the system. But that is not the only advantage. Weak coupling also ensures the composability of software components, because a small communication port increases the probability of finding components that are able to participate in communication.

In Spider, we outline the weakest coupling principle, which is similar to the one used in [21]: "For components to inform other components about anything, they should not address these components explicitly but should send a message with a minimum amount of related data." In Spider, this kind of interobject communication is based on the theory of ADV/ADO [7].

By applying this principle, we expect to obtain maximum composability even when building blocks that have not been planned to work together. There is a good chance that unforeseen combinations of building blocks may occur if the data which has been received or transmitted is simple. The number of combinations may even be greater as the senders or receivers of the information need not be known. In this way the developers (fabricators) of a component have to think only about the input/output ports, whereas the user (builder) of such components may combine them in arbitrary ways, provided that data types match.

In visual languages the workspace becomes too confusing very quickly due to the great numbers of connections and components. To cope with this problem we have introduced two concepts: layers and shadows. The layer concept is very similar to the layer concept used in CAD applications, where the user can break the design in groups of related elements. Shadows are used when we want to show the same component in two different places to minimize the number of connections intersections. We believe these concepts help in the organization of the visual code.

Environment elements may be divided in the following categories:

- Components: they are the main computational elements of the Spider environment. They are black boxes that have ports and properties.
- Component groups: a component group defines the behavior of related components at the design time. It may be argued that the component itself should be responsible for its behavior and appearance at design time. But, we believe the approach we have developed is more flexible, and allows the creation of a component group that handles the behavior and appearance of different components through parametrization. Another advantage of this approach is that it enables the same component to have different design views with different behaviors and appearances. Examples of component groups: database access component group, JavaBean component group, default component group, etc.
- Types: the environment provides a type manager module that doesn't depend on of any particular language. This module reduces the coupling between the composition editor and any particular conventional language. Each type has a list of compatible types that can be used to mirror subtype/subclass concepts found in several conventional languages. Types can be associated with ports or properties. Examples of types: Void, Integer, Rectangle, etc.
- Port groups: a port group helps the fabricator in the definition of composition rules avoiding the creation of several similar rules. Examples of port groups: Action, Event, Attribute, etc.

- Ports and Properties: as mentioned before ports and properties compose the component interface. A port is defined by a name, port group, type, and optional parameters. Each port parameter is defined by a name, port group and type. In practice, however, we can see a port parameter as a subport. Port parameters are particularly useful when defining ports that function as actions or methods. A property is defined by a name and type. We ought to mention that some properties have associated ports. For example, the property Width is associated with the port BoundRectangle.

Examples of ports:

Name	Port Group	Type	Parameters
Show	Action	Void	Message
Hide	Action	Void	-
Click	Event	Void	-
Message	Parameter	String	-
BoundRectangle	Attribute	Rectangle	-
Name	Attribute	String	-

Examples of properties:

Name	Type
BackgroundColor	Color
Width	Integer
Active	Boolean
Label	String

- External Ports: they are the composite component ports that function as bridges to the internal component ports. The ports that are available to composite components are the external ports one creates when defining it.
- Connections: a connection between the ports of two components defines a kind of relationship between them. Examples of connections:
  (Click → Hide)
  (Click → Show (Name → Message))
- Connection Rule: fabricators define connection rules to restrict the component composition so that they certify that only sound connections may be made by builders. A connection rule is defined by the specification of the source and destination port groups. This kind of rule is allowing connections between ports associated with the source port group and ports associated with the destination port group, if the respective port types are compatible. A rule may specify that a connection source and destination are not important (bi-directional connections) so that connections may be made in both directions. Example of composition rules: Event → Action, Attribute ↔ Attribute, etc.
- Dynamic Ports and Properties: this kind of port is created at design time expanding the component interface. We can exemplify the use of these elements with the following example. Supposing we have a component to provide database access. When a database table is selected **at design time** the component will retrieve the database table information and will create dynamic ports to allow the access to the database table row fields.

3. Implementation

The implementation of the Spider environment has been done in two steps. In the first one a prototype has been built using Visual C++ and Visual Basic [17]. This prototype has several restrictions related to the way the architecture has been defined. With the experience of this first step we started to develop the first version of the environment using Java language. The great advantage of this version was the introduction of several domain specific languages (DSLs) [3,9] to define architectures. With these languages the fabricator can define the interface of primitive components, specialize the editor behavior, and coordinate the application generation. The main modules of the version 1.0 may be described in the following way (Figure 2):

- Composition Editor: it is used to make visual composition, the editor behavior is customized for each domain.
- Repository: it is used to store all information used by the environment.
- Application generator: it is used to generate an application created in the Spider environment.

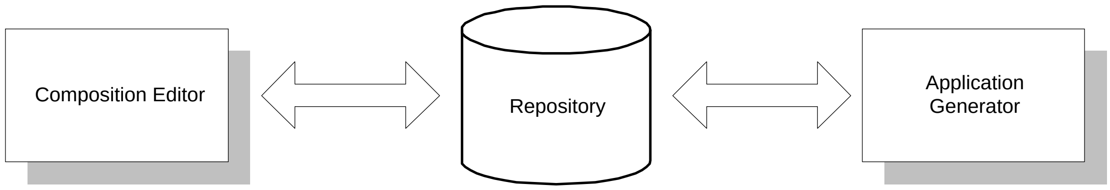

Figure 2 The Spider Environment Big Picture

3.1 Composition Editor

The composition editor is a general purpose graph editor that can be customized for specific domain needs. The editor deals with several concepts (port groups, types, connection rules) that help the fabricator to customize its behavior. We feel that the composition editor design is general enough to handle most of the application domains. As described above, the fabricator may use domain specific languages (DSL) to specialize the editor behavior, but it is also possible to use Java language or the a Script language (section 3.1.2). The advantage of using DSLs is that the description becomes more concise and clear, and there is no need to go into implementation details.

3.1.1 Customizing the Editor Behavior

All the elements of the Spider environment are implemented as Java interfaces that define the elements responsibilities. To simplify the customization process we have provided default implementations (Java classes) for each of these interfaces. To customize the editor, we should provide different implementations for these interfaces. But, we feel that it is not a simple task to customize the editor using this approach. To cope with this problem we allow the customization of the editor using domain specific languages. Two approaches may be used to handle domain specific languages implementation:

- Transform the domain specific language into Java code, that will implement the Java interface.
- Create an implementation that functions as a bridge to the DSL code. In this approach we should implement a DSL interpreter (Figure 3).

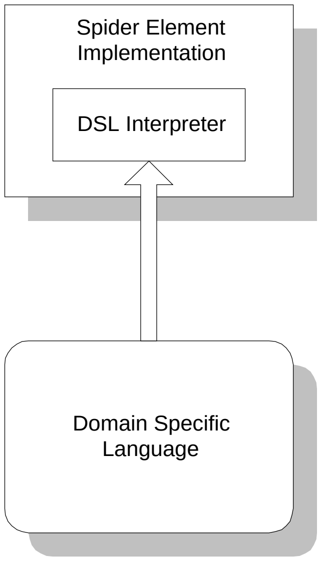

Figure 3 Domain Specific Language Bridge

We have chosen the second approach. It is more dynamic, and there is no need to compile Java code when DSL code is modified. A similar approach has been used to create a bridge to Script code. The Script language (Section 3.1.2) can be used as a starting point in the creation of specialized domain specific languages.

3.1.2 The Script Embedded Language

We built an extension programming language called Script to be used as a flexible configuration language for the Spider environment. This language simplifies the customization work, avoiding the implementation of simple tools in Java language. Another salient point is that the customization process becomes more dynamic, so there is no need to restart the system when a new tool is created or modified. The Script language is implemented in Java language. It is wise to note that being an extension language, Script must be embedded in a Java program (the Spider environment in this case). The Script language features may be summarized in the following way:

- procedural programming
- transparent access to Java code (none stub code is necessary)
- extensible
- reflexive
- interpreted
- support to anonymous and nested functions

It is important to mention that the Script language may be used in other systems, not only in the Spider environment.

3.1.2.1 Script Language Examples

In this section, we show three examples of the Script language code. The two last ones are about building Spider environment tools.

Accessing Java Code

We start by showing how Java objects can be called from the script language. This piece of code writes the following message in the file**arq.txt.**

hello world!
testing script language.
end.

```
FileWriter = class java.io.FileWriter;
PrintWriter = class java.io.PrintWriter;

output = new PrintWriter (new FileWriter ("arq.txt"));
output.println("hello world!");
output.println("testing script language.");
output.println("end.");
output.close();
```

Tool: Align Left

In this example, we describe how a simple environment tool (align left) can be defined in the Script language. Note that the following code uses functions which have been specially defined to accessSpider environment editor information.

```
local firstSelection = getFirstSelectedComponent();
local j = 0;
while (j < getNumSelectedComponents())
{
      local currSelection = getSelectedComponent(j);
      currSelection.x = firstSelection.x;
      j++;
}
```

Tool: Remove All Connections Between the Selected Components

In this more complex example, we describe how a tool to remove the connections between selected elements can be defined.

```
local j = 0;
while (j < getNumSelectedComponents())
{
      local currSelection = getSelectedComponent(j);
      local w = 0
      local numConnections = currSelection.getNumConnections();
      while (w < numConnections())
      {
            local connection =
                  currSelection.getFirstConnectionFrom();
            // verify if is connected to a selected component
            local k = 0;
            local found = false;
            while (k < getNumSelectedComponents() && !found)
            {
                  if (connection.to == getSelectedComponent(k))
                  {
                        connection.remove();
                        found = true;
```

```
                  }
                  k++;
            }
            w++;
      }
      j++;
}
```

3.2 The Application Generator (metaprogramming support)

One of the most important tasks of fabricators is to define how an application is generated starting from the visual code (graphs) built by using the visual builder. The main approach used is transforming the visual code into a conventional textual code. Several systems use string manipulation functions to generate the textual code, but we think that it is a primitive and error-prone technique. Thus, we have developed a metaprogramming system that processes abstract syntax trees (ASTs). By using this system the fabricator may create transformations from the visual code to the desired textual code. The system certifies the syntactical correctness of the generated code. We can compare the metaprogramming system with the Lisp language metaprogramming features.
To create the desired textual code the metaprogramming system provides tree constructors that can be used to generate abstract syntax tree fragments. The tree constructor can compose other AST fragments. The target code is generated by pretty printing the associated AST. We can summarize the process of building an application generator in the following manner:

1. transverse the visual graph code by using tree constructors to build the initial AST fragments.
2. continue to transverse the graph by using the tree constructors to build bigger AST fragments starting from the previously created fragments.
3. repeat step 2 until has been built the complete AST.
4. pretty print the complete AST to convert it into the textual code.

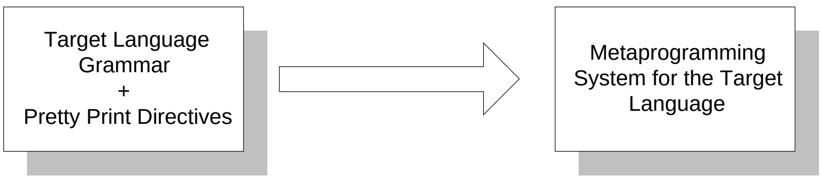

Figure 4 Metaprogramming System Big Picture

To use the metaprogramming system we need to specify the grammar of the target code augmented with pretty printing directives (Figure 4). Each nonterminal found in the grammar specification will be associated with a tree constructor. We can exemplify this process by describing the grammar of a simple language.

```
Block: "{" [NL] [IN] (Statement)* [EX] "}" [NL]
      ;

Statement:  Block |
            "write" Variable ";" [NL] |
            "read" Variable ";"  [NL] |
            …
…
```

We should note that [NL], [IN] and [EX] are pretty printing directives that insert a new line, indent code and extent code respectively. After defining the grammar of the target language, we can use the tree constructors to build the AST fragments. By now we have embedded the tree constructors as a Java language extension to simplify the implementation of the metaprogramming system. In this initial implementation the Antlr 2.11 [24] parser generator tool has been used. The tree constructor extension syntax can be described as:

```
TreeConstructor: “%{” ID “,” TARGET_CODE_TEMPLATE “}%”
      ;
```

Where ID is the name of a nonterminal used in the target language specification, TARGET_CODE_TEMPLATE is a template of code written in the target language. This template can be augmented with references to other AST fragments. We can exemplify the use of the Java extension showing the following example by using the simple language which has been mentioned.

```
// Use the tree constructor Statement to create an AST fragment
Statement stmt1 = %{ Statement, read a; }%;

// Use the tree constructor Block to create an AST fragment that
// has a reference to the AST fragment stmt1
Block blk1 = %{ Block, { %stmt1 read b; write a; } }%;

System.out.println(blk1.prettyPrint());
```

The following code will print the following textual code:

```
read a;
read b;
write a;
```

In the following example a metaprogram that generates Java code is shown. It should be noted that the Java grammar must be specified. In some cases the user can use a subset of the desired target language to avoid the work of specifying its complete grammar.

```
JavaExpression exp = %{ JavaExpression, a < 1 }%;
JavaStatement printMsg = %{ JavaStatement, System.out.println (“Hello
world”));
JavaStatement stmt1 = %{ JavaStatement,  if( %exp){ %printMsg } }%;
JavaMethod method1 = %{ JavaMethod, public static void main
                                    (String args[])
                                  {
                                    int a = 0;
                                    %stmt1
                                  }
                  }%;
JavaClass class1 = %{ JavaClass, public class Test { %method1 } }%;
System.out.println (class1.prettyPrint());
```

The result of this metaprogram will be:

```
public class Test
{
```

```
	public static void main (String args[])
	{
		int a = 0;
		if (a < 1)
		{
			System.out.println("Hello word");
		}
	}
}
```

As described before, the metaprogramming system is embedded in the Java language, but we believe that a minimalist approach is the correct way to be followed in software development. In this minimalist approach one should implement a program by using a more abstract and simpler language that manages to solve our problem. By using this approach program correctness can be proved in an easier way, and also the developer's intentions remain clear and are not blended in a low level language. Thus, we believe that the metaprogramming system must be implemented as a high level language specially designed for metaprogramming.

4. Examples

Let us now see how these concepts are put into use with some examples. These examples are built by using different architectures to show the environment capabilities.

4.1 Shopping List Application

To start with, let us sketch a simple example similar to the one shown in the Visual Age [22] documentation. This example has been implemented by using a general purpose architecture, to create Java applications. The primitive components of this architecture are implemented in Java language, and the target language is Java too. By using this architecture we can compare the Spider Environment with visual programming environments for the Java language as Visual Age for Java [22] and Java Studio [23]. Nevertheless these environments are only general purpose environments, and don't have any domain oriented facility. It is not possible to customize these environments to generate applications in languages other than Java, or to create new abstractions that are specialized for a given domain.
The user interface (Figure 5) of this application consists of two buttons that are used to add and remove elements from a shopping list. The **add** button inserts the **new item** in the **items** list. The **remove** button removes the current selected element from the **item** list.

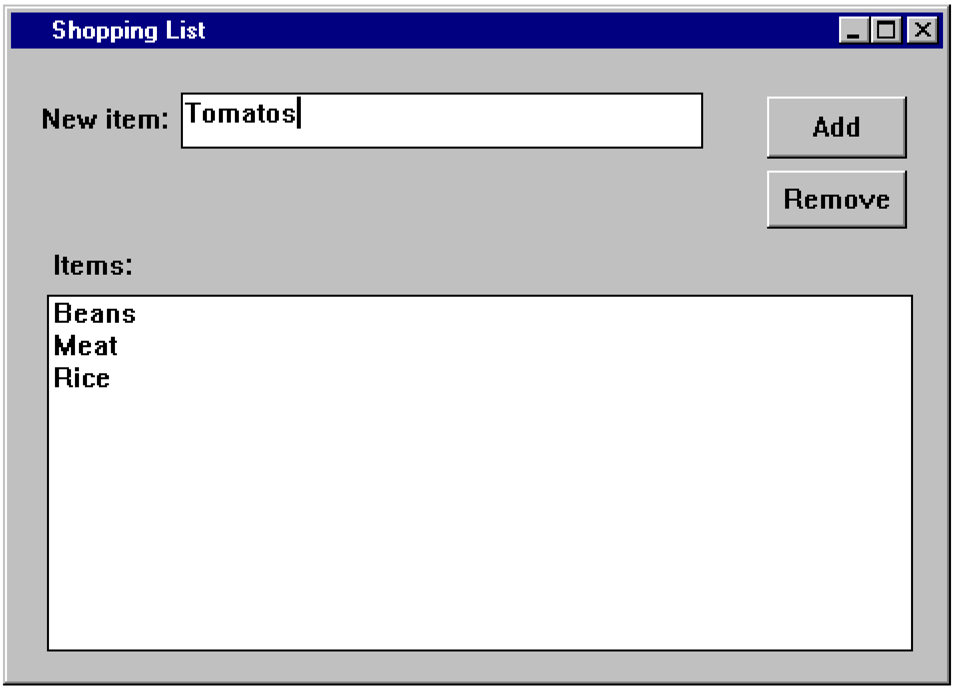

Figure 5 The shopping list application

We start the implementation by creating an user interface component. This kind of component is edited in two separate views in the composition editor: the layout view and the logical view. In the layout view (Figure 6) we can specify the layout properties of the application user interface. The editor behavior in this view is very similar to that of the several existent user interface builders.

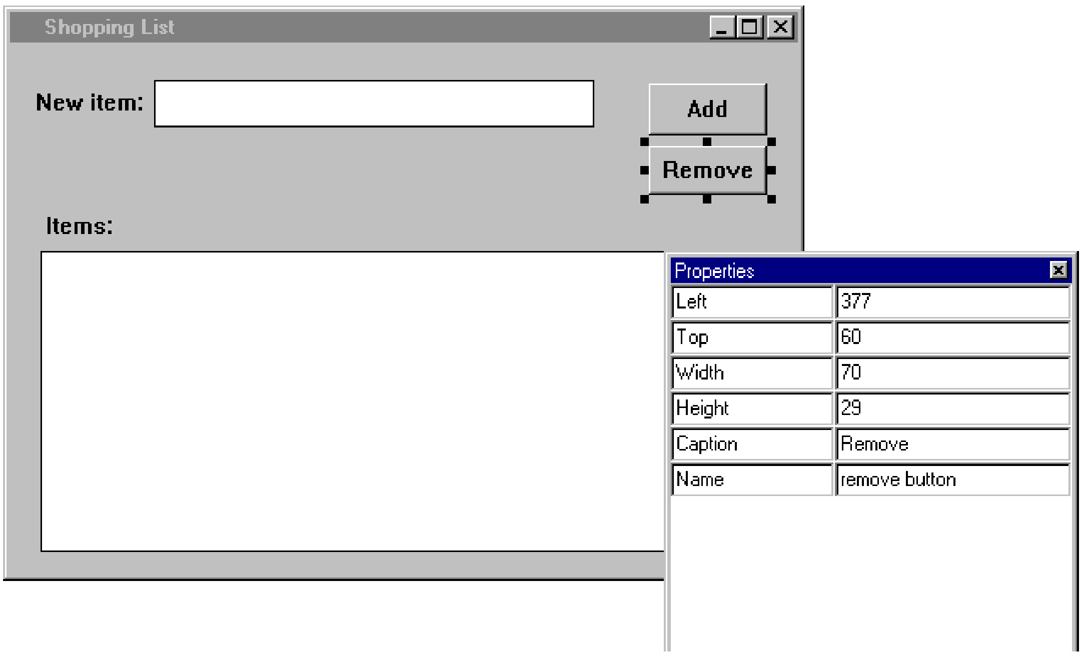

Figure 6 Building the user interface

After specifying the layout properties of the user interface, we can start the specification of the application logic (Figure 7). We implement the desired functionality by making the following connections:

- `(addButton, click) → (shoppingList, addElement`
  `(itemName, text) → (elementToAdd)`
  `)`

  This connection specifies that when the user clicks the add button, the action addElement is executed. The element to be added is specified by the connection with the elementToAdd parameter.
- `(removeButton, click) → (shoppingList, removeSelectedElement)`

  This connection specifies that when the user clicks the remove button, the action removeSelectedElement is executed. We should note that this action doesn't have any parameter.

Finally, the application is built by using the application generator to generate Java code that implements the desired functionality specified in the composition editor.
The architecture used in this example has been used in the construction of the Spider environment, after an initial bootstrap has been made.

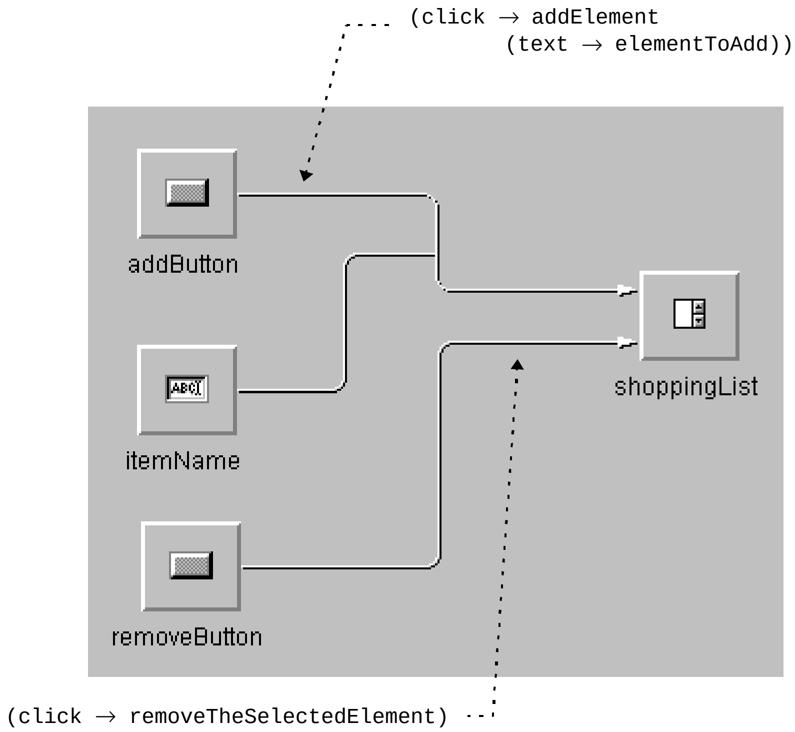

Figure 7 Application logic

4.1.1 Additional Issues in the General Purpose Architecture

When starting to use the general purpose architecture we perceived that the task of mapping Java classes to components was too tiring. To avoid the necessity of this mapping we created a universal Java class component. With this component we only need to select the desired Java class and it will create dynamic ports to allow the access to the public methods and fields. An Action port is created for each class method, with one parameter for each method parameter. The class fields are mapped to attribute

ports. To implement the universal class component we use the Java language reflexive properties that allow the access to class meta-information.
We use a similar approach to create a universal JavaBean class component. But in this case we also map JavaBeans events to component event ports.

4.2 Internet Applications

By now the creation of Internet applications involves several technologies and standards. As examples we would like to mention HTML, Java, JavaScript, VBScript, ActiveX, etc. Therefore to create Internet applications using these technologies becomes a complex and difficult task. We think that this task may be simplified by creating a Spider architecture specially tailored to generate Internet applications.
The first decision we made when designing the architecture was to allow the HTML page layouting in any editor so that the builder can use his favorite HTML editor. The pages created using the HTML editor should be processed by a parser to encapsulate the HTML information inside a Spider component. This parser generates information for the composition editor and the application generator (Figure 8). JavaScript and CGILua [10] can be also encapsulated in components by the parser.
It is not necessary for builders to know JavaScript and CGILua, as we have implemented several primitive components by using these two languages. By using this approach we have created an architecture where builders can be able to build applications even though they may know very little about these technologies.
To detail this architecture we have described two application examples. The first example is a simple application whose main purpose is to show all the steps and details involved in the Internet application generation process. The other application objective is to show the environment flexibility in the creation of complex Internet applications.

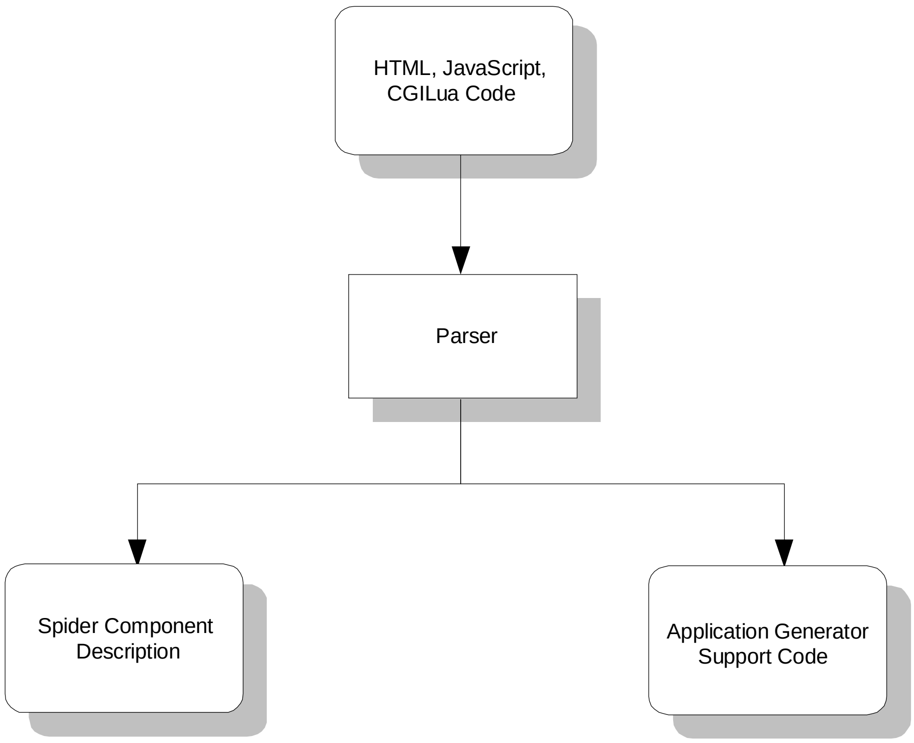

Figure 8 Importing HTML code

4.2.1 Simple Application

In this example, we will describe all the steps that are needed to create an Internet application by using the Spider environment. In this simple application the user can type a message in the entry field, and after that press the **Show** button that opens a message box containing the text typed by the user (Figure 9).

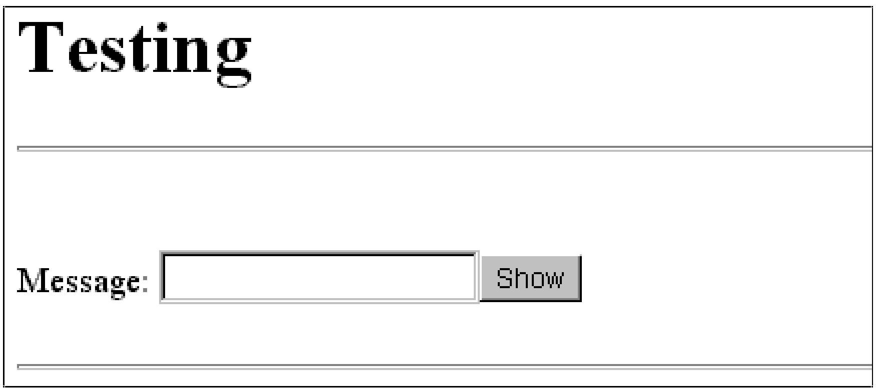

Figure 9 Application Interface

The first step in this simple application is to create the user interface with the editor one prefers. In this example we have used the Netscape Composer. When we creating HTML pages to be used with the Spider environment, it is necessary to augment these pages with information to be used by the parser to be able to generate the dynamic ports and properties. In this example we define two ports: MessageText and ShowBtn.

The first one provides the text typed by the user and the other signals that the button has been clicked. In the following HTML code the definition of these two ports is shown in bold.

```
<HTML>
<HEAD>
   <META HTTP-EQUIV="Content-Type" CONTENT="text/html; charset=iso-8859-1">
   <META NAME="Author" CONTENT="Leonardo Moura">
   <META NAME="GENERATOR" CONTENT="Mozilla/4.02 [en] (Win95; I) [Netscape]">
   <TITLE>Testing</TITLE>
</HEAD>
<BODY>

<H1>
Testing</H1>

<HR WIDTH="100%">

<P><FORM>
<BR>Message: <INPUT   TYPE="text"   NAME=%{FIELD.MessageText}%   SIZE=15   ><INPUT
TYPE="button" VALUE="Show" onClick=%{EVENT.ShowBtn}%>
<BR></FORM>

<P>
<HR WIDTH="100%">
</BODY>
</HTML>
```

Next, we will define the application logic by using visual composition. First we must use the parser to encapsulate the HTML page created in the previous step inside a Spider component. After that we can start to make visual connections. The process may be summarized in the following way:

1. Create a new composite component called SimpleApplication
2. Use the parser to encapsulate the HTML Page inside a Spider component.
3. Add the predefined JS_MessageBox component in the composite component.
4. Connect the port **MessageText** to the port **Message**
5. Connect the port **ShowBtn** to the port **Show**
6. Generate the application

By making these connections we are specifying that when the button **ShowBtn** is pressed the JS_MessageBox **Show** action will be executed and will show the message specified by the connection**MessageText→Message** (Figure 10).

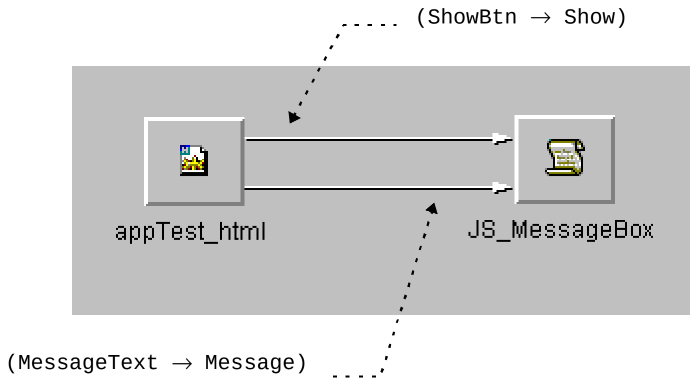

Figure 10 Application Logic

In the step 6, when we generate the application code, the environment will generate a HTML page that contains the application interface and logic. We should mention that the environment is responsible for inserting the code needed to correct implement the desired application functionality. The generated HTML code is shown below.

```
<HTML>
<HEAD>
   <META HTTP-EQUIV="Content-Type" CONTENT="text/html; charset=iso-8859-1">
   <META NAME="Author" CONTENT="Leonardo Moura">
   <META NAME="GENERATOR" CONTENT="Mozilla/4.02 [en] (Win95; I) [Netscape]">
   <TITLE>Testing</TITLE>

<SCRIPT>
<!--- COMPONENT MESSAGEBOX
function MessageBox(message) {
       alert(message)
}
// -->
</SCRIPT>

</HEAD>
<BODY>

<H1>
Testing</H1>

<HR WIDTH="100%">

<P><FORM>
<BR>Message: <INPUT  TYPE="text"  NAME="Message"  SIZE=15  ><INPUT  TYPE="button"
VALUE="Show" onClick="MessageBox(this.form.Message.value)">
<BR></FORM>

<P>
<HR WIDTH="100%">
</BODY>
</HTML>
```

4.2.2 Customer List Application

In this section, we have used the customer list application to describe the creation of Internet application with several components and accessing server databases. The database access has been made by using primitive components implemented in CGILua [10]. We should mention that primitive components written in JavaScript are executed in client machines, nevertheless the components written in CGILua are executed in server machines. This difference between the primitive components are not seen by the builders. They use the primitive components as black boxes where only the connection ports can be seen. We configure the architecture to allow only sound connections between the primitive components so that the CGILua and JavaScript code don't call each other.

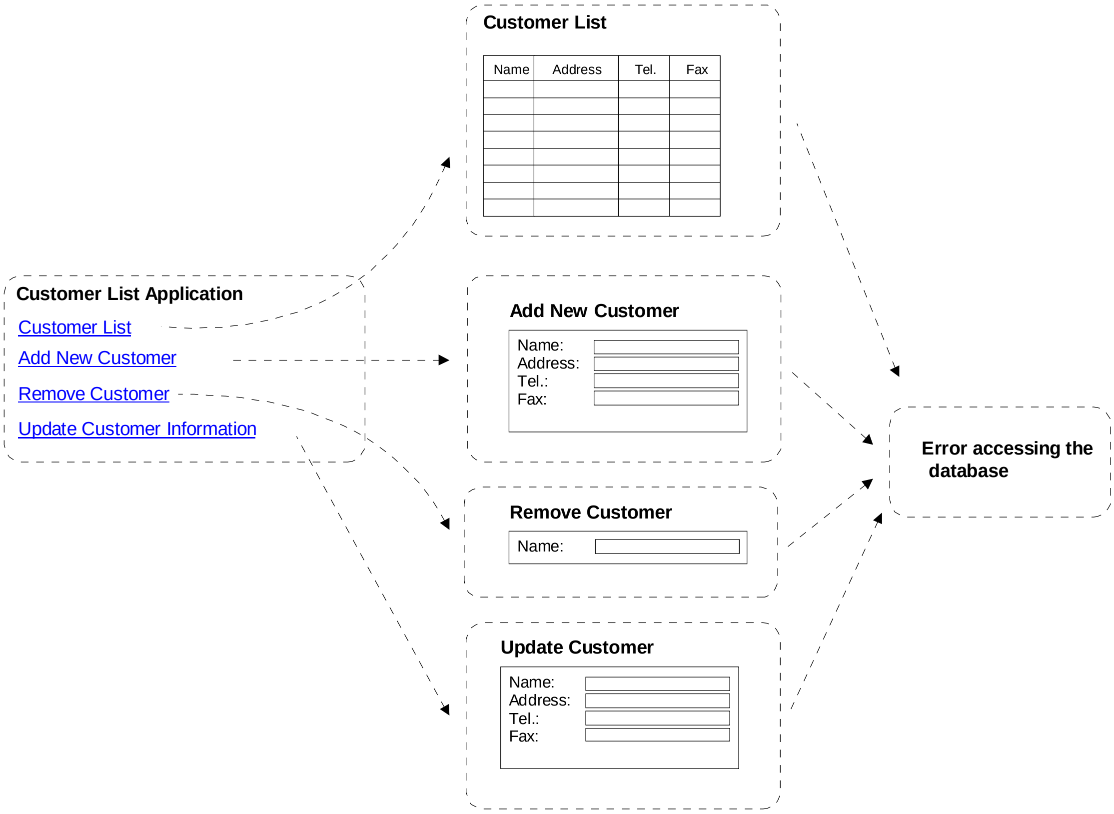

Figure 11 Customer List Application

Let us start by using a HTML editor to create six HTML application pages. Later on we can use the parser described in the section 4.2 to encapsulate the pages in Spider components. The parser will encapsulate each HTML page in a different component. To have a modular application we should create five composite components:

- Composite Component - Add New Customer: implements the application logic to add a new customer. To implement this component, we select a database access component, which allows the access to the usual database functions (insertion, deletion and the updating of rows). The component design is shown in Figure 12. Note that this component defines an external port called **open**, which allows other components to open the Add New Customer HTML page. The component behavior can be summarized in the following way:

    - When the user presses the button **addButton**, the action **add** of the database access component is executed and the information provided in the entry fields to fill the new customer information is used.
    - If there is an error in the action execution, the event **error** is thrown by the database access component. This event will open an error page.

- Composite Component - Customer List: implements the application logic to list information about all the customers stored on the database. The implementation of this component is very similar to the previous one.

- Composite Component - Remove Customer: implements the application logic to remove a particular customer.
- Composite Component - Update Customer: implements the application logic to update information about a particular customer.
- Composite Component - Main: implements the application main page using the four components described above. In this component we should note that when the user selects the HTML link in the main page, the component associated with the link is activated (Figure 13).

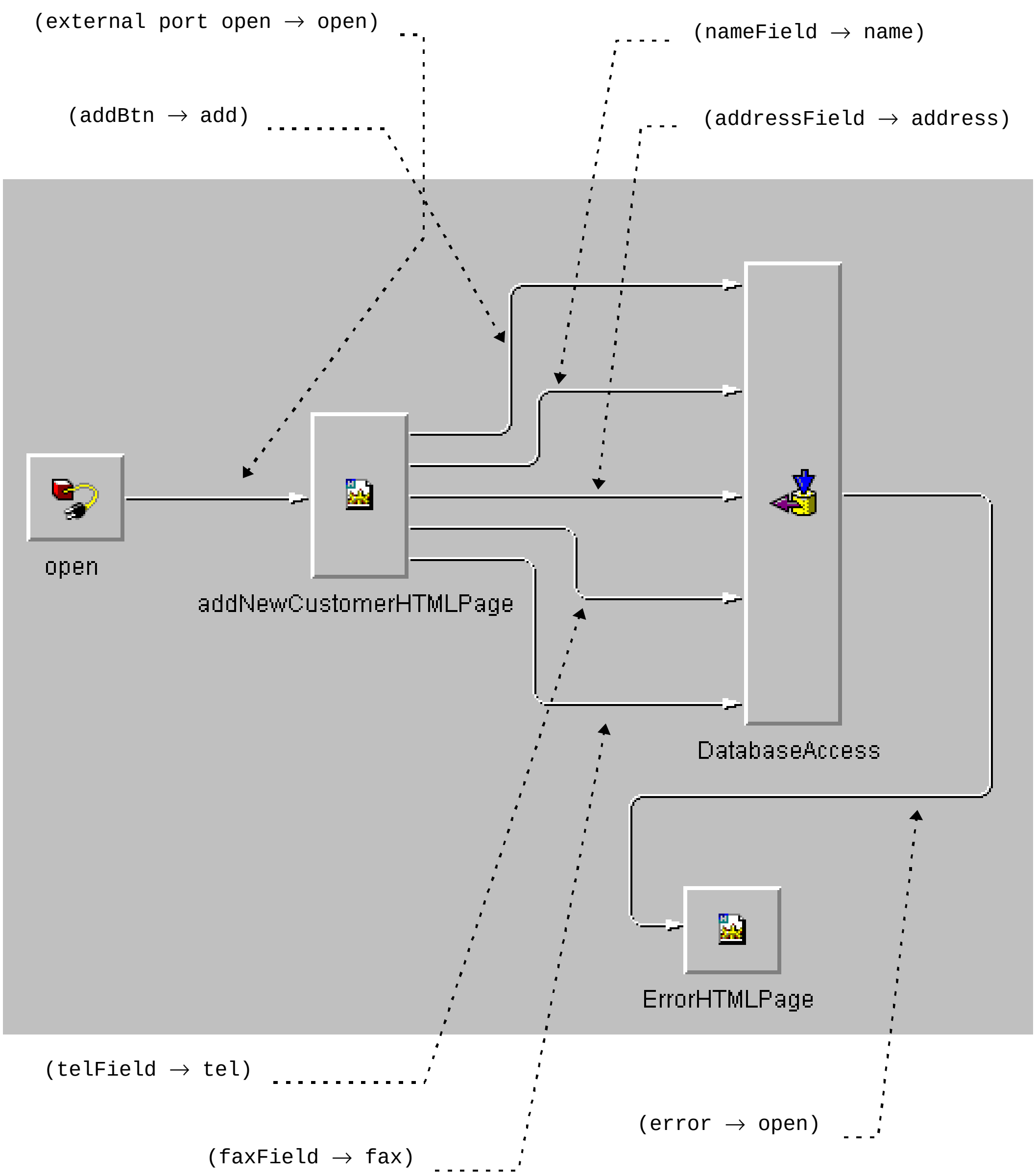

Figure 12 Composite Component - Create New Customer

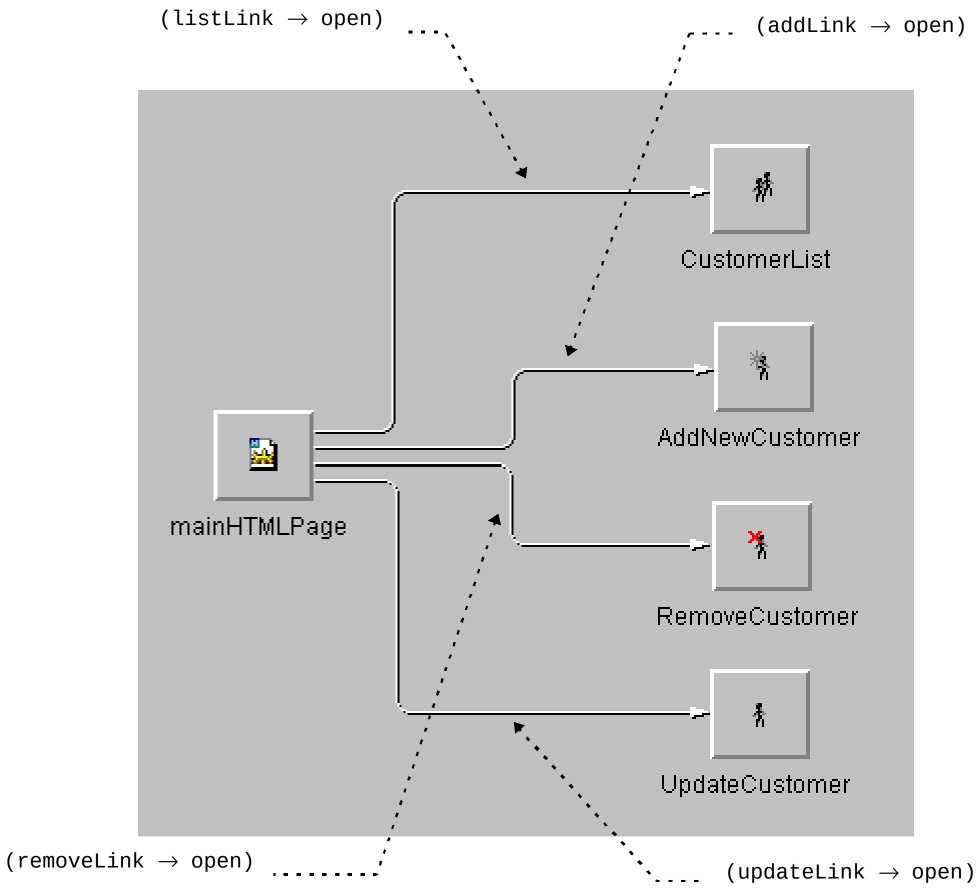

Figure 13 The Main Composite Component

4.3 Transformation Systems

Transformation Systems [4,11] are applications that modify a program or specification code. The main functions of these modifications is to improve code performance; to transform code which is in one language into another one; to generate implementation code starting from specification code; to apply reverse engineering processes; to extract code information; and so forth. The HTML parser defined in section 4.2 is a good example of the use of transformational systems. The operation of most transformation systems can be summarized in the following way (Figure 14): first the program or specification code is parsed to generate an abstract syntax tree (AST); next, the abstract syntax tree is modified by rewriting rules; finally, the abstract syntax tree is converted into code again by using the pretty printer module.

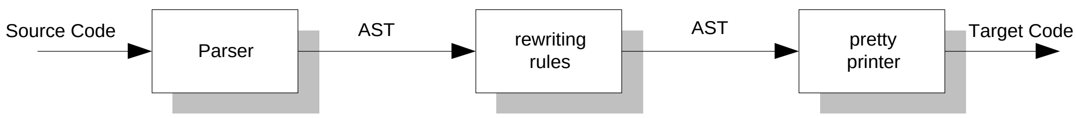

Figure 14 Transformation Process

The steps involved in the transformation system process suggest a data flow paradigm that can be easily mapped in the Spider environment. To implement this mapping we have created a Spider architecture which has been specially tailored to build transformations. The first step when building this architecture was to identify its primitive components. We feel that the following components are the basic building blocks in this architecture:

- Parser: this component transform textual code into an abstract syntax tree. The component is customized by a BNF code that specifies the grammar of the source code.
- Match and Apply: rewriting rules are separated in two kinds of components: match and apply. The match component searches the abstract syntax tree for a given pattern. The apply component creates a new abstract syntax tree fragment starting from other fragments. These two kinds of components are customized by search and building patterns.
- Pretty Printer: transforms the abstract syntax tree into target code. This kind of component is customized by using a language which is similar to the one described in the section 3.2 for the metaprogramming system.

We believe that this architecture introduces reuse, component and modularization components into transformation definitions. These concepts are not found in most of the transformation systems.

4.4 Web Based Education

The Web Based Education architecture is a specialization of the Internet architecture. This architecture is part of a large project in the software engineering laboratory (LES) at PUC-Rio. The main objective of this project is to build an environment in with Web courses may be developed. This environment will allow teachers, who have little experience in using Internet, to build courses with a WWW interface.

At the moment, we are working on ways of identifying and building architecture components. We are applying the experience acquired in Internet architecture. The component identification process will be made by searching for design patterns [8] that are usually found in web courses which have been created with Internet architecture. This process may be summarized in the following way:

1. creation of several web courses using Internet architecture.
2. identification of design patterns in the web courses created in the previous step.
3. architecture specialization and creation of new components specially tailored for web courses development.

We feel that it is an interesting and productive approach, as it allows professionals from other areas to build applications in their own domain. For instance, Internet architecture may be specialized for HomeBanking and InternetShopping domains.

5. Integrating different architectures

When large systems are been developed it is not rarely that these systems are extended to several application domains. Therefore, it is necessary to have some mechanism to integrate components created in different architectures. To make this integration we have been using component wrappers that function as bridges between different domains (Figure 15). We think that this approach is feasible due to the following features of the environment:

- The component description encapsulates the language or architecture used to implement it. We may mention that this component description plays a role similar to the IDL language of the CORBA standard.
- The application generator metaprogramming features allow fabricators to generate all the stub code (component wrappers) needed to connect components created in different languages or architectures.

For instance, we may want to add Java applets to the HTML pages created with Internet architecture, but Java applets are created in the general purpose architecture. Therefore, we must use a component wrapper in Internet architecture to access the Java applets created in the general purpose architecture.

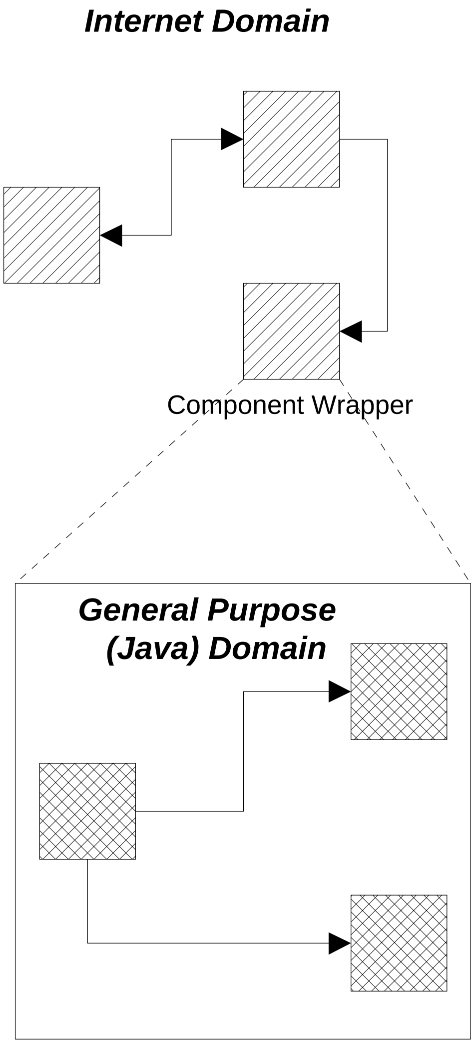

Figure 15 Integrating different architectures

6. Related Work

As the Spider environment can be used in the implementation of different kinds of applications, it is not an easy task to compare it with other environments and tools. As it is an environment for visual composition, it may be compared to VisualAge and Vista environments.

Vista [20, 12] is a visual composition tool that is part of the ITHACA project [1, 13]. The main differences between Vista an the Spider environment are related to: the construction of primitive components in Vista are a hard task and can only be made by using C++ language; Vista doesn't explore advanced visual concepts such as layers and shadows.

Visual Age [22] is the first successful commercial product that uses visual composition to develop applications. The main differences between VisualAge and the Spider environment are: in VisualAge, the developer must know the language (C++, Java or Smalltalk) where the components were built very well; and the components built in one language can't be used with components built in others; VisualAge is not domain oriented, it can only be used as a general purpose environment. The Visual Age composition paradigm is driven to enable the connection between user interface objects to the application logic objects. Thus, we believe that in the VisualAge environment we can only build simple applications by using only visual programming.

As the Spider environment is an extensible environment that can be customized for different application domains, we can compare it to Escalante [15]. Escalante gives support to the interactive design, prototype building, and automatic generation of visual language environments. This system allows developers to specify an application and its interface by the definition of the data and visualization models. The definition is supported by a visual specification environment. The data model is composed by generic graphs, and the visualization model is composed by graphics. After the definition of the data and visualization models, the Escalante environment will generate a program that implements the data model and the visualization mechanisms. The resulting program can be improved by the insertion of textual code. This environment has interesting mechanisms to define visual languages that are easy to be used, but doesn't give any support when adding new components or reusing a specific application code.

7. Current / Future Work

At the moment we are improving and building new architectures. In the future, we plan to investigate other ways of integrating different architectures.

We also intend to create an architecture to define new architectures. We may call this architecture a "meta-architecture". Thus, the fabricator will build new architectures by using visual composition. The use of this "meta-architecture" joined to Transformation architecture will make it much easier to create new architectures.

8. Conclusion

We believe that by combining the visual composition paradigm with a domain oriented approach we may have taken a step in the right direction. The visual composition paradigm provides a graphical interface for the interactive application construction by regular developers. Highly specialized developers are architecture (environment) builders that create custom solutions for application domains that can be used by less specialized developers. It no longer makes sense to develop large monolithic applications by using general purpose languages that blend the programmer's intention inside implementation details. The various resources available to customize the composition editor make it unnecessary to create different editors for different domains.

The metaprogramming features used in the application code generation allow the generation of a very efficient code. Furthermore these features manage to separate the high level abstractions from implementation details in the correct manner. Fabricators can customize the code generation by using particular optimizations which are valid in the domain.
The architectures described in the section 4 show that this technology can be used to allow the creation of applications for a great number of users. Architectures which are specially tailored for a given application domain make it feasible for professionals of other areas to build applications in their own domain. We believe that the Spider environment can make programming less frustrating and more productive.

9. Acknowledgment

The authors would like to thank Bruno Bondarovsky, Luidi Fortunato, Marcelo Sant'Anna, Pedro Ripper, Philippe Fremau for giving their support in the implementation of the Spider environment. We are also grateful to the National Research Council of Brazil (CNPq).

10. References

[1] Ader, M.; Nierstrasz, O.M.; McMahon, S.; Miller, G.; Profrock, K.; "The ITHACA Technology: A Landscape for Object-Oriented Application Development"; Proceedings, Esprit 1990 Conference; pp. 31-51; Kluwer Academic Publishers;Dordrecht; 1990.

[2] Arango, A.; "Domain Analysis Methods"; Software Reusability; Harwood; London; March; 1993.

[3] Bell, J.; et al; "Software design for reliability and reuse: A proof-of-concept demonstration"; In TRI-Ada '94 Proceedings; pp. 396-404; ACM; November 1994.

[4] Cordy, J., Carmichael, I., "The TXL Programming Language Syntax and Informal Semantics", Technical Report, Queen's University at Kinkston, Canada, 1993.

[5] Shu, C. N.; "Visual Programming"; Van Nostrand Reinhold; New York; NY. 1988

[6] Clements, P.; "From Domain Model to Architectures"; A. Abd-Allah et al., eds.; Focused Workshop on Software Architecture; pp. 404-420; 1994.

[7] Cowan, D. D.; Lucena, C. J. P.; "Abstract Data Views: An Interface Specification Concept to Enhance Design for Reuse"; IEEE Transactions on Software Engineering; March 1995.

[8] Gamma, E.; Helm, R.; Johnson, R. E.; Vlissides, J.; "Design Patterns, Elements of Reusable Object-Oriented Software";Addsion-Wesley; 1995.

[9] Hudak, P.; "Building Domain-Specific Embedded Languages"; Computing Surveys; 28A(4); ACM; 1996.

[10] Iersalimschy, R.; Borges, R.; Hester, A. M.; "CGILua A Multi-Paradigmatic Tool for Creating Dynamic WWW Pages"; SBES 97 (Simpósio Brasileiro de Engenharia de Software); 1997.

[11] Leite, J. C. S.P.; Sant'anna M. ; Freitas, F. G.; "Draco-PUC: a Technology Assembly for Domain Oriented Software Development"; Proceedings of the 3rd IEEE International Conference of Software Reuse; 1994.

[12] Mey, V.; "Visual Composition of Software Applications"; Ph.D. thesis (no. 2660); Dept. of Computer Science; University of Geneva; 1994.

[13] Mey, V.; Nierstrasz, O.; "The ITHACA Application Development Environment"; Visual Objects; pp. 267-280; D. Tsichritzis (Ed.); Centre Universitaire d'Informatique; University of Geneva; 1993.

[14] Meyer, B.; "Object-Oriented Software Construction"; Prentice Hall International; 1988.

[15] McWhirter, J. D.; Nutt G. J.; "Escalante: An Environment for Rapid Construction of Visual Language Application"; Technical Report CU-CS-692-93; Department of Computer Science; University of Colorado; 1993.

[16] Moura, L. M.; Braga, C. O.; Lucena, C. J. P.; "Instantiating Frameworks with Domain Specific Languages"; to appear.

[17] Moura, L. M.; Lucena, C. J. P.; "Uma Introdução ao Ambiente Visual Spider"; SBES 97 (Simpósio Brasileiro de Engenharia de Software); 1997.

[18] Moura, L. M.; Lucena, C. J. P.; "O Ambiente Visual Spider para o Desenvolvimento de Aplicações para a Internet"; to appear.

[19] Moura, L. M.; "Uma Linguagem de Programação Visual"; Tese de Mestrado; Departamento de Informática; PUC-Rio; 1996.

[20] Nierstrasz, O.; Gibbs, S.; Tsichritzis, D.; "Component-Oriented Software Development"; Communications of the ACM; 35(9); pp. 160-165; 1992.

[21] Schiffer, S.; Fröhlich, J. H.; "Visual Programming and Software Engineering with Vista"; Visual Object-Oriented Programming;Prentice Hall; 1995.

[22] Visual Age User's Guide and Reference; IBM Inc.; 1997.

[23] Java Studio Reference, Sun Microsystems, 1997.

[24] Parr, T.; Lilley, J.; "ANTLR 2.10 Reference Manual"; MageLang Institute; http://www.MageLang.com