Business Component Prototyper for SanFrancisco: An experiment in architecture for application development tools

by H. van Emde Boas-Lubsen

The technology behind distributed processing, frameworks, and the Java language environment is relatively new and changing fast. This requires that current application development tools adapt more quickly to requirements of their users. This paper describes how these tools can be made more flexible and customizable. As an example, the architecture of Business Component Prototyper for IBM SanFrancisco™ is presented. In its current form, Business Component Prototyper is a tool to develop prototype applications provided with IBM SanFrancisco v 1.4. Its objective is to help new SanFrancisco developers create small prototypes using the SanFrancisco foundation layer programming model, without requiring the heavy tool set used for production application development. IBM SanFrancisco is a Java™-based set of components that allows developers to assemble server-side business applications from existing parts, rather than build "from scratch."

The use of object technology for application development is slowly becoming mainstream. Despite this, the promise the technology seemed to hold—an increase in productivity by an order of magnitude—has not yet been fulfilled, as we demonstrate later.

We start to recognize that more is needed than just object-oriented languages and object-oriented analysis methods. Standardization, ready-to-use components, and development environments where major parts of the code are generated for the developer should solve the problems. Standardization is easier if all developers use the same programming language. A large segment of the information technology community has decided that the Java** programming language and specifications for JavaBeans** and Enterprise JavaBeans** will bring this much needed unity. As a communication protocol for documents and structured data, XML (Extensible Markup Language) is emerging. Based on these standards, serious component frameworks are starting to appear, such as IBM's SanFrancisco*.

Are components the solution?

As a pioneer of component technology, Brad Cox[1] should be mentioned as the inventor of the Software-

 0018-8670/00/$5.00 © 2000 IBM

IC**. It seems that more than ten years after he published his seminal book, his ideas finally have become a reality.

David Taylor[2] has an interesting argument why CASE (computer-assisted software engineering) tools are not the solution. Assume a traditional development effort where equal development time is spent on analysis, design, code, test, and maintenance. Consider then the perfect object-oriented CASE tool, which would make analysis and design 30 percent more effective, reduce implementation to zero, and make testing and maintenance 30 percent faster. All together, our total development effort would be improved by less than 50 percent, which is far from the order of magnitude improvement we require.

Another well-publicized approach is *reuse*. Edward Yourdon[3] has a good argument for why reuse has not been very successful in the past. One of the reasons is that most reuse was *code* reuse. Design or analysis reuse would be more effective, because it would automatically include code reuse, and it would improve our previous calculation for productivity considerably.

We can achieve *design reuse* by employing one of the component architectures that are currently defined, such as the following:

- Sun's JavaBeans and Enterprise JavaBeans specifications
- ActiveX**, VBX (Visual Basic** extensions), and OCX (OLE [object linking and embedding] Custom Controls) specifications from Microsoft
- IBM's SanFrancisco application business components (in Java code)

Component frameworks have characteristics of both code and design reuse. They promise to make reuse easier by providing not only well-structured and well-bounded pieces of functionality as building blocks, but also the means to glue these building blocks together quickly and easily.

We conclude that component frameworks will help to improve productivity considerably. If accompanied by the proper development tools, productivity could be even higher.

SanFrancisco

A very large component building effort is the IBM SanFrancisco project. It is unique because it includes not only middleware, but also a large base of business components. It is outside the scope of this paper to give an in-depth overview of SanFrancisco—please see http://www.software.ibm.com/ad/sanfrancisco for more information. For readers unfamiliar with the framework, this section contains a very short description of its layered structure.

As shown in Figure 1, SanFrancisco delivers three layers of code reusable by application developers:

- The foundation layer provides the infrastructure and services that are required to build applications in a distributed, multiplatform environment.
- The common business objects layer provides implementations of frequently used business objects that are common to more than one domain. The common business objects can also be used as a base for interoperability between applications.
- The core business process layer provides business objects and default business logic for selected "vertical domains." SanFrancisco currently delivers business components in the domains of accounts receivable (AR), accounts payable (AP), general ledger (GL), order management (sales and purchase), and warehouse management.

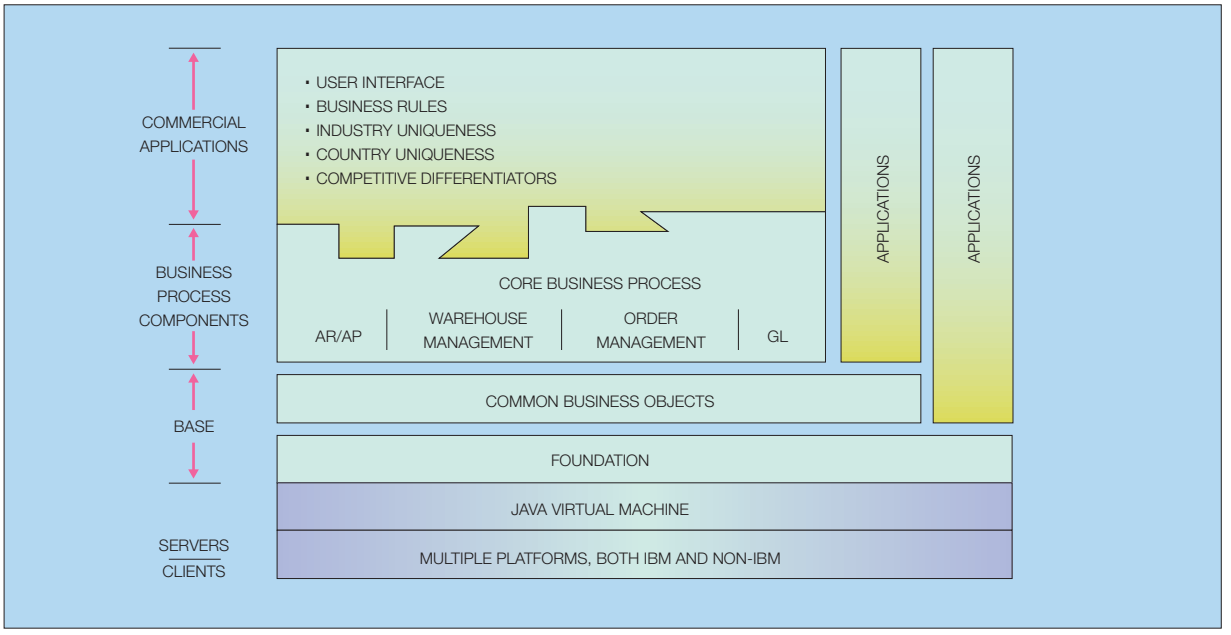
Why do we need additional tools?

Most major modeling tool and IDE (integrated development environment) vendors introduced new versions of their tools to accommodate the Java programming language and UML, the Unified Modeling Language.[4]

Many tool vendors find it difficult to adapt their tools to large frameworks, for various reasons:

- Some modeling and development tools do not scale very well when more than 2000 components (the number of components that SanFrancisco provides) have to be imported.
- Code generation technology does not scale well for such a large programming model as SanFrancisco defines.
- The gap between modeling and programming tools is still wide.
- Most of the tools are not yet written in the Java language. Therefore they often run on only a single platform and have fixed functionality.
- Support for components as "black-box" building blocks is lacking.
- It is often impossible or very difficult to adapt code

Figure 1 Overview of SanFrancisco



generators to specific framework or user requirements.

Because of these problems, good tools targeted at the SanFrancisco framework were slow to appear and some that are there now still lack functionality.

It seems that there is room for a new tool architecture. This architecture should utilize components in the same way that it tries to support them. It should exploit new document facilities and use them to store model information, program source code, and design documentation.

Business Component Prototyper

Business Component (BC) Prototyper is a prototyping tool provided with SanFrancisco v 1.4.[5] Its objective is to help developers new to SanFrancisco to understand the foundation layer programming model more easily. It allows them to experiment by creating prototypes that can actually run. It is positioned within the SanFrancisco product as an evaluation tool, because it allows developers to use the *evaluation* version of SanFrancisco. In this version the Rational Rose** models of SanFrancisco classes and the SanFrancisco code generator are not available.

The version of BC Prototyper described in this paper has more functionality than the released tool, most notably the facility to make use of SanFrancisco components. This version is not available for general use.

At first sight, BC Prototyper is a simple integrated development environment encompassing a range of activities that span from object modeling to application deployment. It includes a modeling tool, a programming environment, a code generator, a GUI (graphical user interface) builder, and SanFrancisco utilities. A closer look reveals a building-block architecture, which enables fast adaptation to changing requirements. The architecture utilizes the ideas behind new developments such as Java *beans*, Java *introspection* and XML. BC Prototyper was used extensively to develop itself.

The Business Component Prototyper is a pure Java tool that can run on any client platform supported by SanFrancisco. It is a simple CASE tool, and it also has some integrated development environment ca-

pabilities, code generation facilities, and interfaces to many SanFrancisco utility functions.

Overview. We first give an overview of the functionality of BC Prototyper in its SanFrancisco configuration. Next, we describe BC Prototyper's flexibility features and how the tool can be used in other environments.

The core part of the tool, the code and user interface, are generated from a model of the tool itself. By changing this metamodel (using BC Prototyper), this part of the tool can be changed.

The functionality of BC Prototyper can be expanded by writing plug-in adapters, which adhere to a simple bean-like interface.

The code generator accesses user model information through Java introspection into the metamodel objects. The code generation templates are coded in XML, and therefore are easily customizable by the user. New coding templates can be added by the user.

BC Prototyper has a process to create *components* from any Java `.class` file. As an example, BC Prototyper components have been made for most IBM SanFrancisco common business objects. They can now be used in BC Prototyper in a black-box fashion. Loading large models into the tool can be avoided in this way.

Often-used SanFrancisco utility functions are available via tool-bar button plug-ins. This enables fast and easy iteration through the SanFrancisco development cycle.

Tool functionality. Before we describe the architecture of BC Prototyper, it will be helpful to describe its functionality. We do this by showing a simple SanFrancisco application, developed with BC Prototyper.

In this application a company owns many address books, one for each employee. Each address book can contain a number of addresses. AddressBook is a class we must create in our model; Address is a component taken from the SanFrancisco common business objects. Other classes in the model are AddressBookController and DescribableDynamicEntity. The Controller class provides a mechanism for the company to own the address books in a tightly coupled collection. Maybe one would expect a Company class in the model; it is not visible, because it is handled implicitly by BC Prototyper. In general, Company objects play a central role in any SanFrancisco application; they can be arranged in a complex, hierarchical structure and serve as anchor points for all application data. The DescribableDynamicEntity class is a subclass of the Entity class, and therefore objects of its subclasses (AddressBook and AddressBookController) are automatically persistent, transactable, and distributable. In addition, objects of the DescribableDynamicEntity class and its subclasses can be described in a language-independent way, and attributes can be added to them dynamically. Finally, the Distinguishable interface allows AddressBook instances to have identifiers (IDs), which can be used to look them up in the AddressBookController instance.

When the tool is started, a window will appear with three parts (see Figure 2). The first part contains a *menu bar* and *tool bar*. The tool-bar buttons show the main actions a user can perform in the SanFrancisco configuration of the tool:

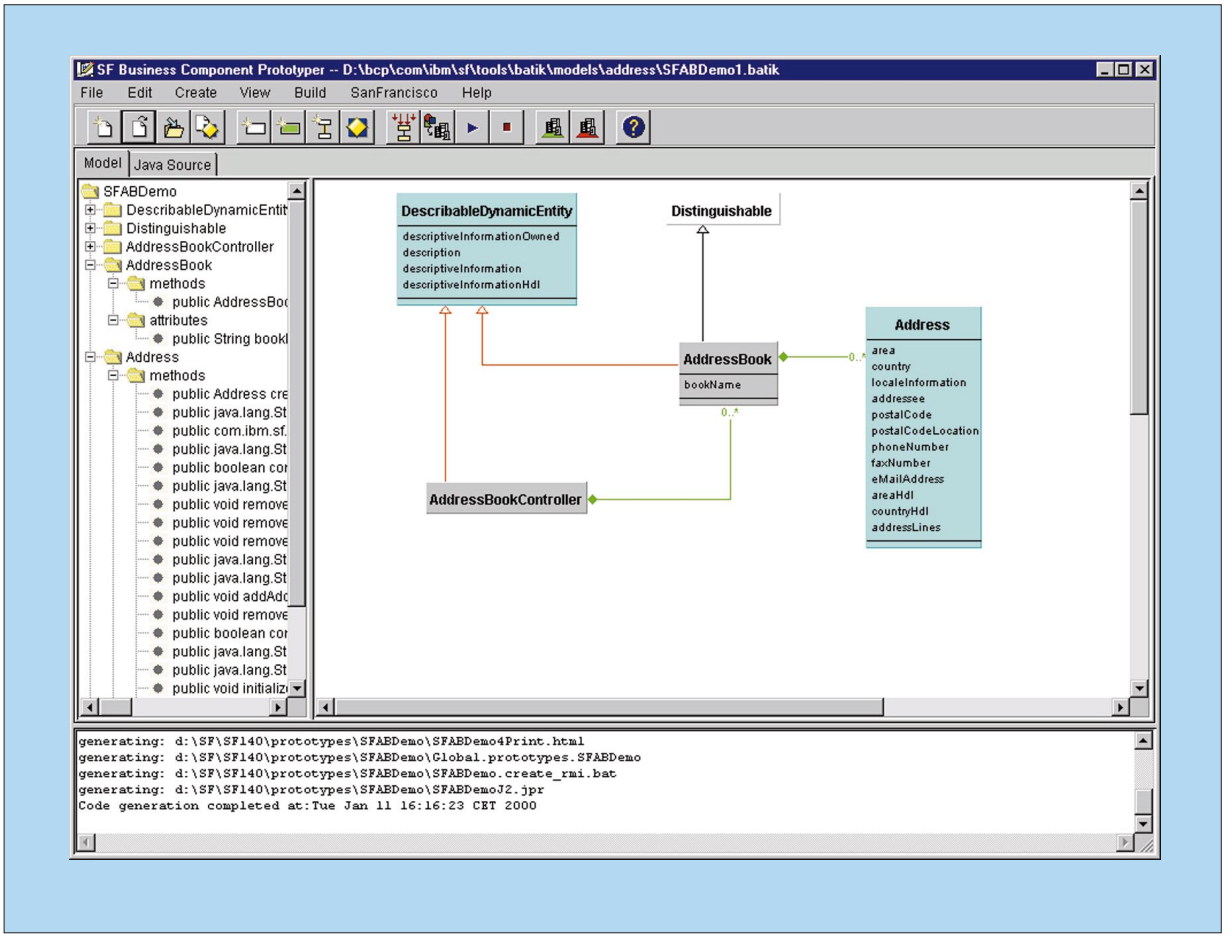
- Create new project
- Load and save project
- Edit project properties
- Create new class
- Import component
- Create relationship
- Edit selected model items (classes or relationships)
- Perform build for selected components (generate server, client, and GUI code, compile, generate proxies)
- Append names to SanFrancisco name space
- Run the application
- Stop the application
- Start SanFrancisco
- Stop SanFrancisco
- Show help information

The second part is a work area, with one or more tab panels.

The first panel contains two subpanels. The left subpanel lets the user view the model in tree form. Model elements can be updated by double clicking the mouse button, after which a property editor appears. The right subpanel displays the currently loaded model in graphical form.

The left side of the second panel shows a list of `.java` files found in the package directory of the current project. With the buttons provided, the files can be shown in a simple text editor, or compiled with the `javac` compiler. For any of the `.java` files

Figure 2 The BC Prototyper window



listed on the right side of the second panel, a proxy can be generated. The generated naming information, needed by the SanFrancisco distributed object infrastructure, can also be viewed and updated in this panel.

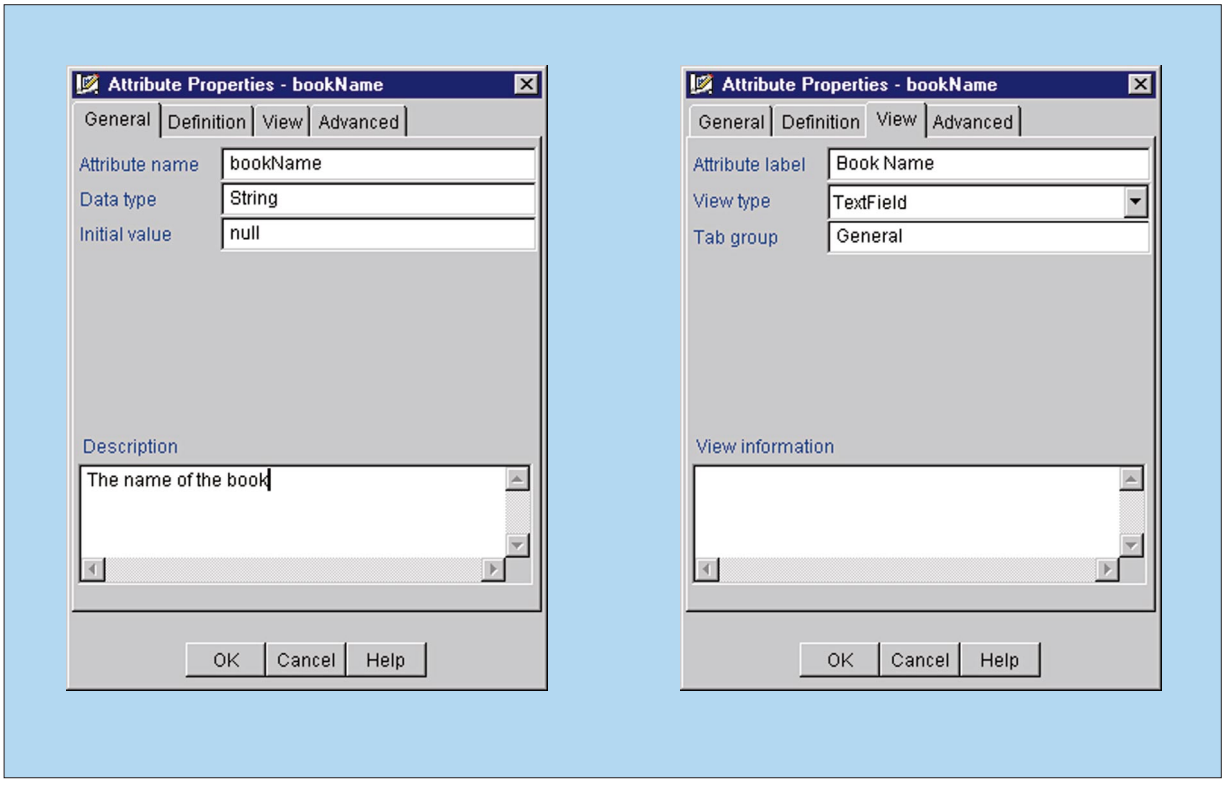
The third part is a message area, where the actions the tool performs are logged and errors are displayed.

This paper is not a BC Prototyper users' guide, therefore we do not describe in detail how to create the application. We mention a few points that are different from other tools. The graphics of the model are a simplified form of UML. Anyone capable of writing a Java drawing routine could replace the graphics module with one that does a better UML job.

The imported components are shown as green class shapes. In Figure 2 these are *DescribableDynamicEntity* and *Address*. A tool-bar button can be pressed to start the import. The components are kept in a simple file structure. A file dialog lets the user choose the desired component. How components are converted to BC Prototyper format from SanFrancisco classes is described later.

The white class shape shows the *Distinguishable* interface. Interfaces are imported in the same way as components. Additionally, interfaces can contain code generation fragments. This allows default implementations to be provided for a Java interface. The developer usually does not have to write any code to implement the interface.

Figure 3 Property editor for attributes



The tree view shows details of the model. Updates can be made by double-clicking on one of the items in the tree, which causes a property editor to appear.

Noteworthy also is the opportunity to customize the GUI code that is generated by BC Prototyper. As an example, we show the property editor for the *bookName* attribute in the AddressBook class (in Figure 3). In the example it is specified that *bookName* will be represented as a text field, and the label will be "bookName." It is possible to change the view type and to set the label shown to the left of the text field to something else as the attribute name, for example "Book Name." The code generator would then generate the appropriate code.

Note that the property editor shown is itself an example of view customization. It consists entirely of code generated from the tool metamodel, described later.

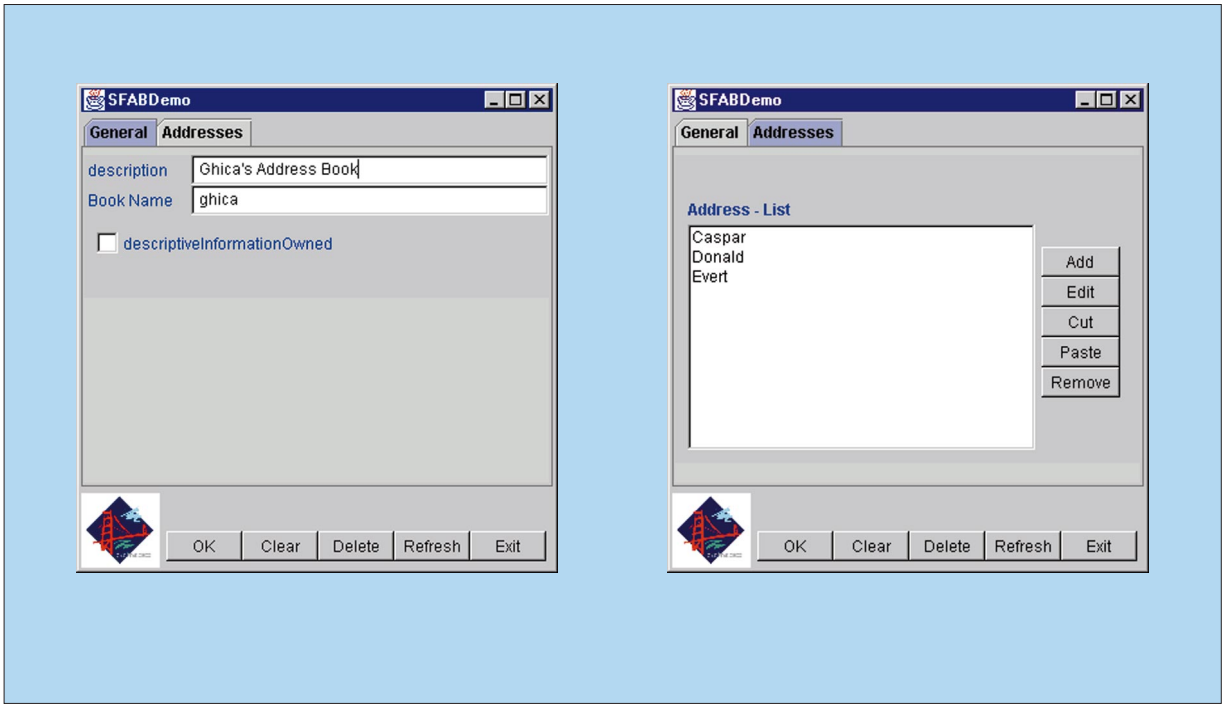
When the model is completed, pressing the "Rebuild" button on the tool bar generates and compiles the code and generates auxiliary Java classes needed by the distributed object architecture of SanFrancisco. The next step is to add the name information tokens for the newly defined classes to the SanFrancisco naming configuration, using a utility. The SanFrancisco servers will automatically be started if necessary.

Next, the application can be started. Figure 4 shows a screen image. Note that the fields in the "Addresses" panel are all generated from the information in the imported component.

The architecture of BC Prototyper

The basic structure of BC Prototyper is very simple: the *toolbase* is the spine of the tool (see Figure 5). It keeps a reference to the metamodel structure and a list of references to plug-ins. The only processing the toolbase performs is to notify all plug-ins of events. Each plug-in can choose to ignore or to react to the event, possibly generating further events.

Figure 4 The generated main window for the address book



Action plug-ins are made visible as tool-bar buttons or menu items. *Tab panel* plug-ins encapsulate, as the name suggests, tab panels. Available for use by the plug-ins is a set of tools, for example, drawing routines and the code generator. Plug-ins are activated by clicking on the toolbar and choosing a menu item, or by notification from the toolbase.

An `.ini` file lists the plug-ins that should be loaded at tool startup. By varying this list, the functionality and appearance of the tool can be modified. For example, leaving out the SanFrancisco-related plug-ins would provide a stand-alone prototyping tool.

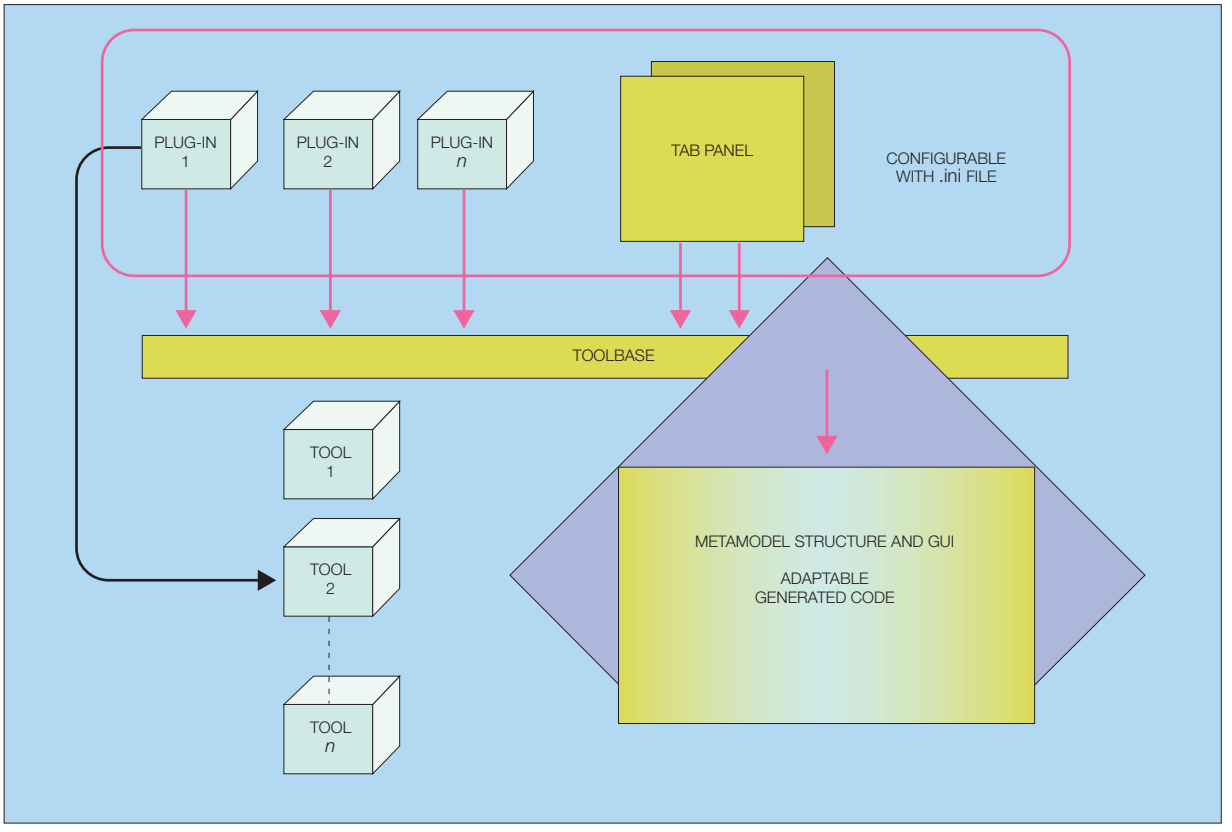
The metamodel. A *metamodel* is a model of a model. To clarify: when a model is defined in a modeling tool such as BC Prototyper (or Rational Rose), the user defines what the classes are in the model, then, for each class the user defines several characteristics, such as class name, etc. For each class, the user also defines its attributes, its methods, and its relationships to other classes.

In the same way that one can build a model for the address book application, where the classes are AddressBook and Address, one can also build a model from the information BC Prototyper uses to keep information about models. In the BC Prototyper model the classes are: ClassHeader, ClassBody, Attribute, Method, etc. as shown in Figure 6. We call this model the *metamodel* of BC Prototyper.

As Figure 6 demonstrates, the BC Prototyper metamodel is a model. Like other models, code can be generated from it. We have done that, and this code now plays a central role in the tool itself. It is used to keep the information about the models *users* define, and the generated GUI code provides the user interface to the tool. See, for example, the property sheets in Figure 3.

Again, because the metamodel is like any other model, it can be changed like any other model, although it is not advisable to change the structure or the class names. Code can be regenerated and compiled from the model. When BC Prototyper is restarted, any changes will be reflected in the GUI, and values in any new attributes will guide the code generation of new applications.

Figure 5 The structure of BC Prototyper



Looking closely at the metamodel information in Figure 6, and at Table 1, we find that many items are apparently there to support the SanFrancisco framework. By using another metamodel, other environments could be supported such as an Enterprise JavaBeans environment or simple stand-alone applications.

As an example of the details to be found in the metamodel, Table 1 shows the (generated) documentation for the attributes in the *Attribute* class.

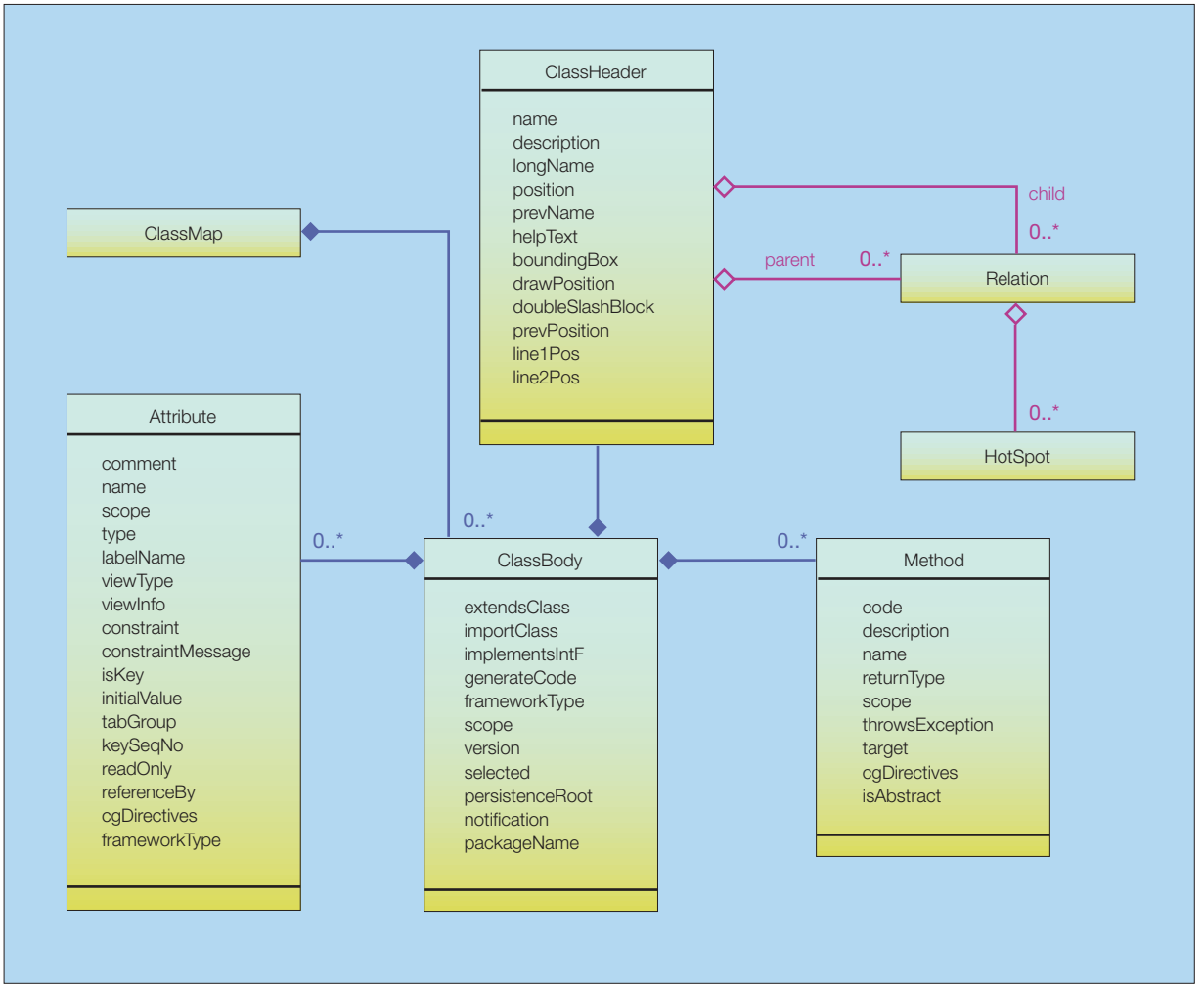
Mapping from model to view. BC Prototyper generates, if requested by the user, not only model code that implements the structure of the model as defined by the user, but also a GUI that is useful for prototyping and that can be used for later customization into a production version of the application.

In Figure 7 we show a transport order management example. In Figure 8, we show part of the generated user interface. This example shows that even for a more complex model it is possible to generate a *usable* interface. When we found problems in one of the many examples we tried, the cause for strange-looking interfaces was always a mistake in the model.

The mapping rules for creating the user interface are listed here:

- A class maps to a *Frame* instance with a set of tab panels.
- A class attribute maps to a *TextField*, *TextArea*, *Checkbox*, *Choice*, or *Button* instance, with a *Label* instance if appropriate.
- A one-to-many contained relationship in a class will map to a *List* instance in the containing class. The list item shows the result of the `toString()` method applied to the contained object.
- For a class with a contained one-to-one relationship, the attributes will map to tab panels within

Figure 6 The BC Prototyper metamodel (redrawn from BC Prototyper screen image)



the frame of the containing class. Superclasses are mapped in the same way.

- For other than containing relationships, a reference button (labeled "Details") will be added to the parent class.
- Attributes and lists can be grouped across the tab panels by specifying a *TabGroup* value in the view tab of the attribute or relation property editor.

We claim that mapping by these rules will always result in a usable GUI. The reason for this is that a class should define a *coherent set of concepts*, and on a window on a screen, we want to find values of a coherent set of concepts displayed. Therefore, either a one-to-one mapping can be automatically generated, or something is not correctly defined in the model.

For attributes, a reasonable mapping is applied by default. For example, a String attribute will map to a TextField instance, and a Boolean attribute will map to a CheckBox instance. This mapping can be changed through the property sheet for an attribute by defining a different view type value. See Figure 3 for an example.

Applying these rules to our transportation model, we obtain a generated frame for the TransportOrder class as shown in Figure 8.[6] The way the class attributes are assigned to groups, where a group maps

Table 1 Generated documentation for the Attribute class in the metamodel

GUI Label	Description	Attribute
Description	Purpose of this instance attribute	comment
Attribute name	Name of the attribute	name
Scope	Scope of the attribute	scope
Data type	Attribute type—default is String. Also viewable are "String," "char," "Boolean," "int," "byte," "short," "long," "float," and "double."	type
Attribute label	Name used as label for a text field, etc.	labelName
View type	How the attribute will be shown in the GUI—default is TextField. Also supported are TextArea, Choice, and Checkbox. If Choice is specified, the strings shown as choices must be available in viewInfo.	viewType
View information	Contains more information about how the view should appear. Choice option strings can be specified here.	viewInfo
Constraint	A constraint on this attribute. It should be a valid Java expression, evaluating to true or false.	constraint
Constraint exception message	An exception thrown when the constraint evaluates to false.	constraintMessage
This is the key attribute of this object	Indicates whether the attribute is a key attribute for the object	isKey
Initial value	Value given to this attribute when the object is created. It should be a valid Java expression.	initialValue
Tab group	A group name for this attribute. Text fields with the same group name are displayed on a tabbed panel, and the group name is the tab text.	tabGroup
Key sequence number	If this attribute is a key attribute, the sequence number indicates the ordering of the key values. All key attributes together should indicate a unique value in the collection of objects of this class.	keySeqNo
Is read-only	Indicates whether this attribute is read-only	readOnly
Multiplicity	Allows the attribute to be a collection or an array of attributes of a primitive type	referenceBy
CG directives	Code generation directives	cgDirectives
Framework type	Support for attributes with type subclassing Entity, Dependent, or Command; for example, DCurrencyValue (a subclass of Dependent). This avoids the need to have these as actual classes on the canvas.	frameworkType

to a tab panel, cannot be seen from the model as displayed in Figure 7. For example, if one were to open the relationship between "TransportOrder" and "LogisticsUnit," one would find at the view specification that the relationship should be put in the tab group "Goods."

For a production application, the default code generation may not be satisfactory. The tool provides two ways to adapt the generated code to specific requirements:

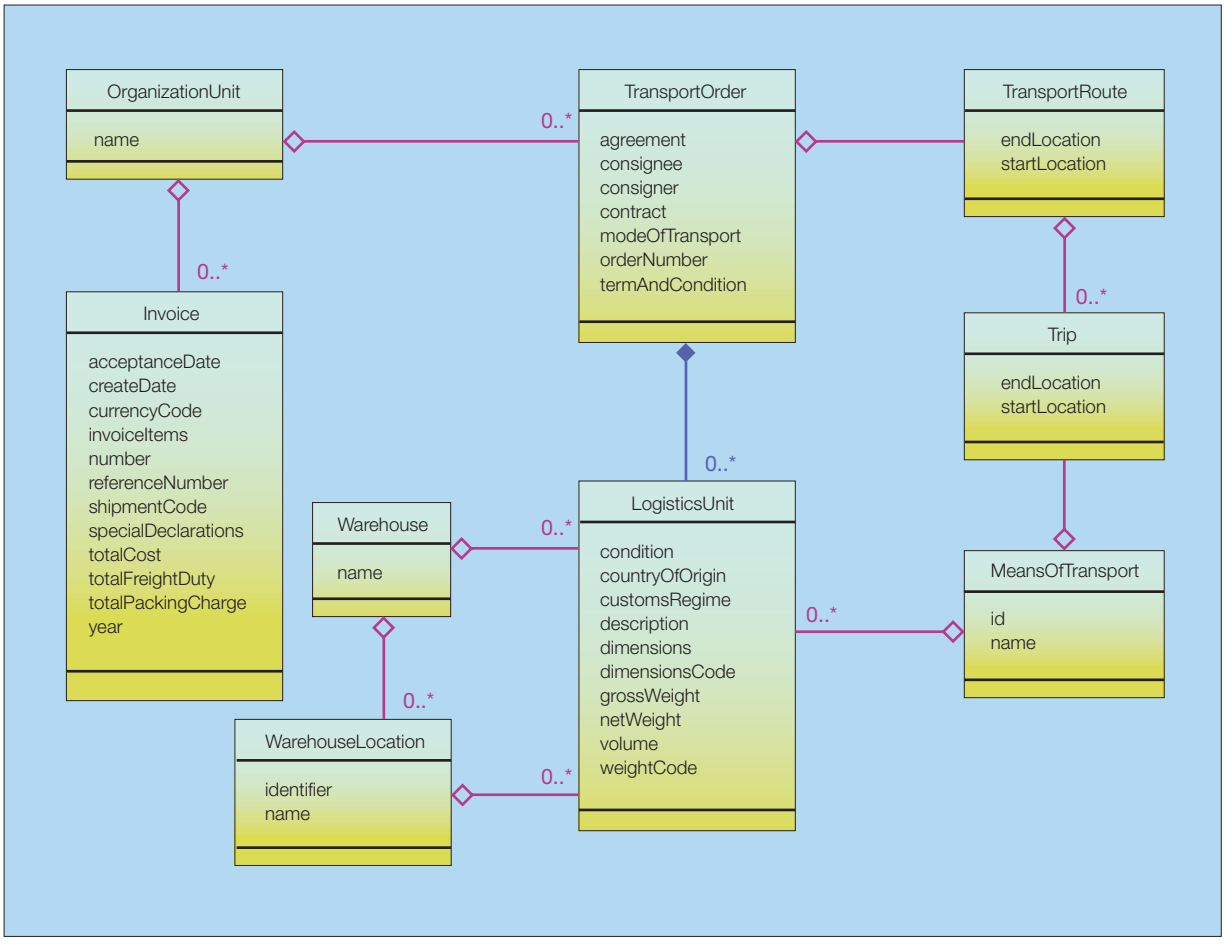
1. Use one of the generated hooks, or override one of the layout methods.
2. Change the code generation templates.

If neither of these is practical for some reason, the generated panels can be imported into a tool with visual editing facilities, such as IBM's VisualAge* for Java, to further extend the code.

Plug-in adapters. The functionality of BC Prototyper can be expanded by writing plug-in adapters that adhere to a simple bean-like interface.

Plug-in adapters perform specific functions, such as opening or saving a model, appending naming in-

Figure 7 A simple transport order management mode (redrawn from BC Prototyper screen image)



formation to the SanFrancisco name space, drawing the model in simplified UML on a canvas, etc.

At startup, the BC Prototyper startup module reads the configuration file and tries to instantiate the plug-ins listed. A plug-in can specify how it is made to be visible during its initialization through simple statements such as:

```
setUsedAsToolbarButton(true);
setUsedAsMenuItem(true);
```

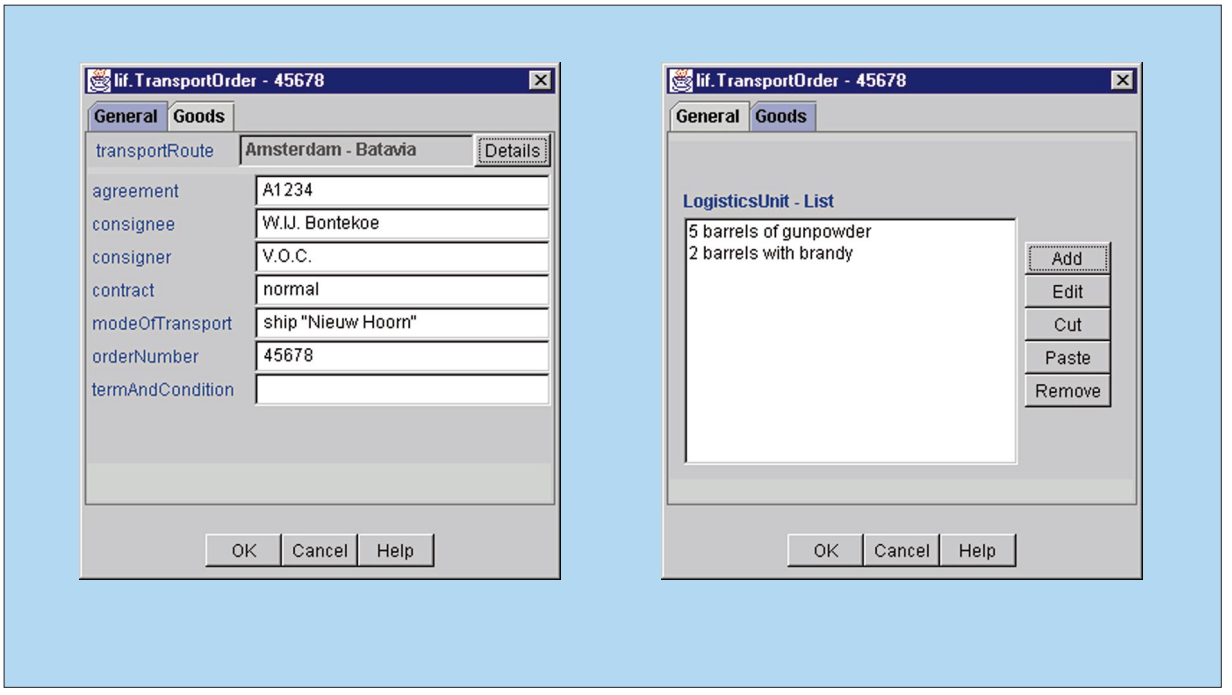
These statements mean that the plug-in will be used both as a tool-bar button and as a menu item. Each plug-in also implements a `run(String s)` method, which will be triggered when the user presses the tool-bar button or clicks on a tab panel, etc.

All plug-ins have a reference to the toolbase. This allows a plug-in to notify the toolbase of *events* to which other plug-ins can react, if appropriate. For example, a plug-in could ask the toolbase to broadcast a *changed* event. All plug-ins will receive the notification and either perform an appropriate action or ignore the event.

Plug-ins are triggered by events to perform a single action and are usually not aware of each other's existence. This means that plug-ins are small and modular. Examples of plug-in functionality were discussed earlier.

The toolbase reference can also be used to access the metamodel structure, described earlier. This

Figure 8 Mapping of the TransportOrder class to a frame with tab panels



structure contains information about the UML model currently loaded in memory.

The modularity of the plug-ins makes it easy to attach diverse functionality to the tool. For example, the plug-ins to start and stop SanFrancisco, which appear as tool-bar buttons in the tool, are totally unrelated to the BC Prototyper functionality. They are just 20-line Java programs that call the appropriate SanFrancisco start or stop program. They become available to the BC Prototyper user when their names are listed in the `.ini` file.

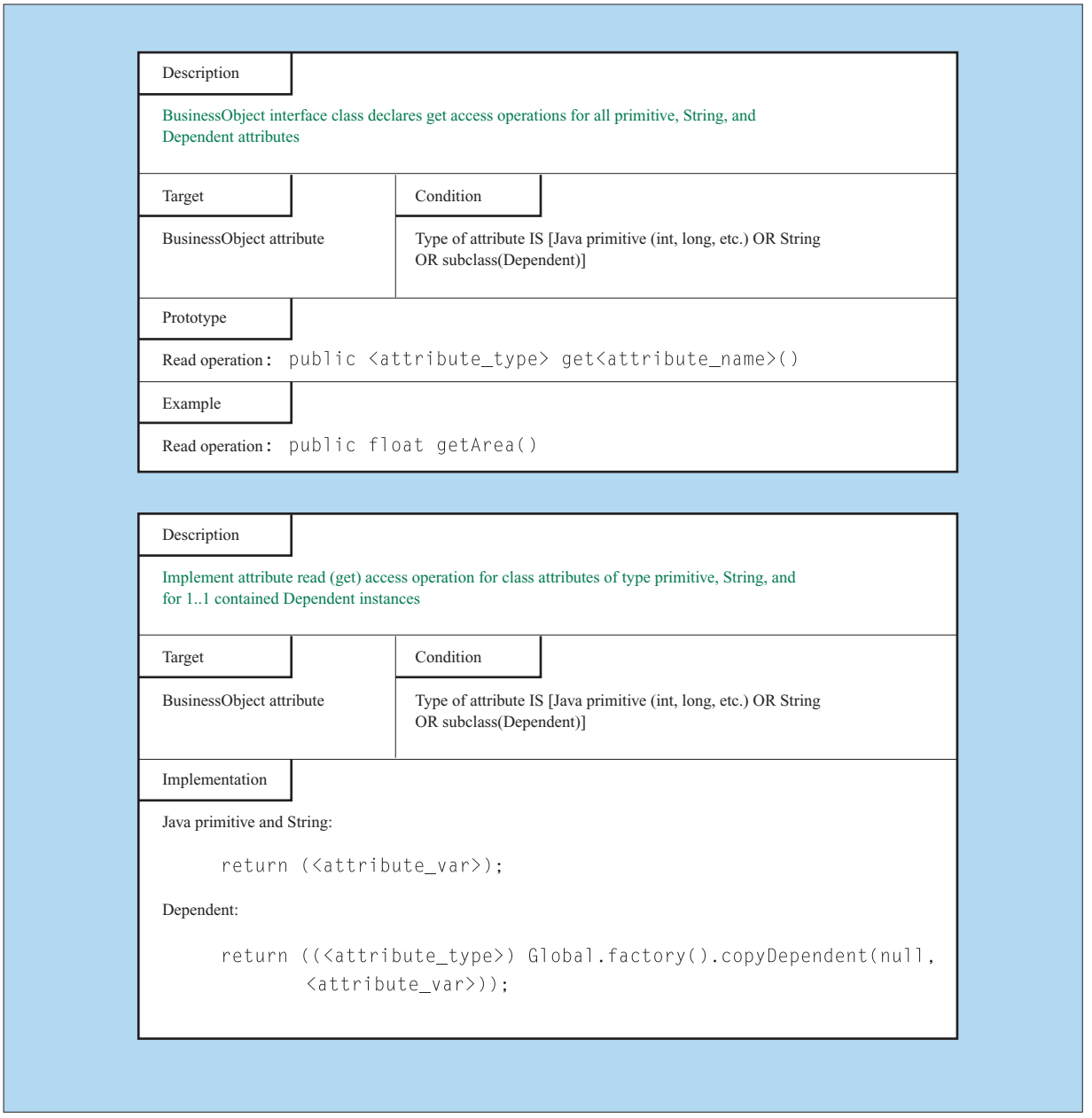
The code generator. Generating code for a framework such as IBM SanFrancisco turned out to be a challenge. The first attempt at writing a code generator proved to be totally inadequate. It was a Java program that examined the model and wrote concatenated strings, interspersed with values from metamodel attributes, to a file. Many code generators use a similar approach, generally using a form of BASIC as a scripting language.

The server-side programming model of SanFrancisco is not very complex, but it is large in size. Client code for transaction management and the GUI adds considerably to the complexity of the code. For our experiment in architecture, a significantly more efficient approach was needed in order to develop a suitable code generator for SanFrancisco.

Objectives for code generation. The introspection-template method for code generation that is used now in Business Component Prototyper fulfills all the requirements on our wish list:

- A template-based approach that allows change without recompiling a (Java) program
- Easy-to-read templates, to allow the developer to easily see what the resulting code will be, and to allow users to adapt the code to special requirements
- Metamodel changes available immediately in the templates
- Java code generation as well as generation of text for multiple purposes, such as HTML documentation, SanFrancisco configuration information, etc.
- Insertion of new templates, for example, to generate code for other frameworks. The new Enterprise JavaBeans standard is an area of interest here.

Figure 9 A simplified SanFrancisco code generation rule



Introspection templates. BC Prototyper uses *introspection templates* for code generation. Introspection templates are XML documents. Each template corresponds to a file to be generated. For example, the SanFrancisco programming model prescribes that, for the AddressBook class in the model in Figure 2, three Java classes (in three `.java` files) must be generated: `AddressBook.java`, `AddressBookImpl.java`, and `AddressBookFactory.java`. For each of these files there is a separate template. Additional templates specify the code for the client implementation.

Figure 10 A snippet of an introspection template

```
<Rule>
<Target>Attributes</Target>
<Condition> type = String OR
            type = int OR
            type = float OR
            type = long </Condition>
     public &type; get&u.name;() {
          return iv&u.name;;
     }
</Rule>
<Rule>
<Target>RelationsAsParent</Target>
<Condition> type = containsOne AND child.frameworkType = Dependent
</Condition>
     public &child; get&u.child;() {
          return (&child;) Global.factory().copyDependent(null,
     iv&u.child;);
     }
</Rule>
```

Another way of looking at templates is to consider them as source code, where XML tags or XML entities (not to be confused with SanFrancisco entities) indicate how this source code should be modified to become the generated code.

The unit for generating code is defined by <Rule> . . . </Rule>.

A rule can have a <Target> . . . </Target> specification, which defines the set of things on which this rule operates. This can be all classes in a model, all attributes for a class, all methods in a class, or all relations of a class.

A target can be restricted by a <Condition> . . . </Condition> element. For example, in the rule in Figure 10, only attributes of types `String`, `int`, `float`, or `long` are considered.

Finally, a rule can contain prototypical source code and XML entities in the source code. XML entities are delimited by "&" and ";". For example "&type;" in a rule will be replaced by the actual type of the attribute. During code generation, these XML entities are replaced by values that are found by introspection into the model.

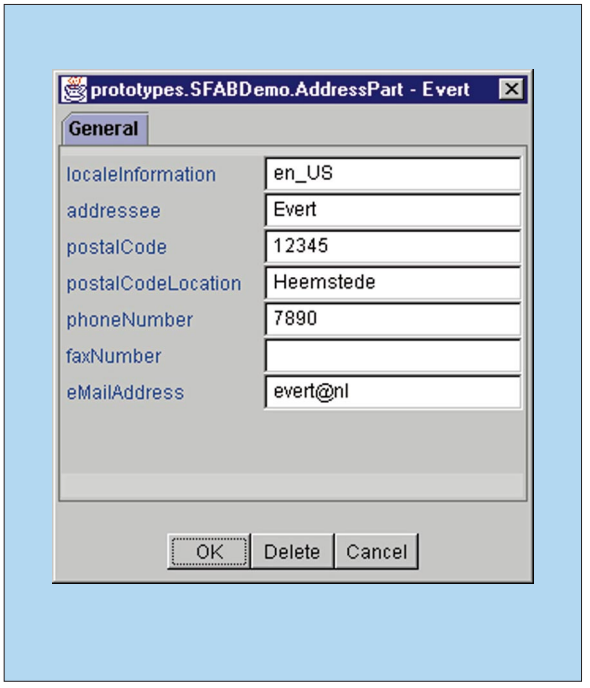
This sounds very abstract, and we need a concrete example. Let us look at a set of rules in the SanFrancisco programming model that define how to implement a "get access" operation for primitive or String attributes and one-to-one related Dependent classes. Figure 9 shows a simplified form of the rules.

The translation of the rules is not completely straightforward, because the way SanFrancisco defines the generation rules in the programming model does not easily map onto the BC Prototyper metamodel. The BC Prototyper template is shown in Figure 10.

The generated code for the AddressBook class, which has *one* attribute, bookName, of type String (causing the first rule to be evaluated once), and *no* relationships to single Dependent instances (causing the second rule to be skipped), would be:

```
public String getBookName() {
     return ivBookName;
}
```

Figure 11 The generated GUI for the *Address* common business object



XML entities require special processing, and we are not sure yet whether this can be made to comply with the XML standard.[7] The reason is that, if we use parsed XML entities, the DTD (data type definition) must change during processing and be adapted to current values in the model.

An introspection template has a tight relationship with the metamodel structure. The XML entity names correspond with the attribute names of the classes in the metamodel. During processing of the template, the XML entities are replaced by the corresponding attribute values.

Using introspection templates, the speed of *developing* a code generator is completely determined by the availability of suitable example code. It was possible to develop a code generator that generates code with functionality almost comparable to the standard SanFrancisco code generator in a short time. In addition, BC Prototyper completely generates client code and the GUI. This client code performs transaction management and services, suitable for maintenance tasks on the business objects, and includes a GUI. Also, some useful `.bat` files, and files containing the SanFrancisco naming information, etc., are generated.

The quality and performance of the generated application is, of course, completely determined by the *content* of the templates. Because templates are so easy to change, we found that we also could generate better quality code.

Use of components in BC Prototyper

Any development tool that supports a framework, like IBM SanFrancisco, should have a strategy to support components. When modeling in Rational Rose, the developer simply needs the full SanFrancisco model available and models the new application on top of it. For BC Prototyper, that did not seem like a very good idea. It would mean that only components for which a UML model is available could be used. For the SanFrancisco framework this is true, but using this approach would make BC Prototyper top-heavy and not scalable.

Another option is to use the source code importer of BC Prototyper. We considered using it to re-engineer components from source files. The main disadvantage of this approach is that source is not always available.

The solution chosen for BC Prototyper was to import information from `.class` files. This is a very general solution and can be applied to any compiled Java file. For SanFrancisco, some of the programming model patterns have been applied in the re-engineering process. It was found that some information cannot be extracted fully. The usability of many SanFrancisco tools could improve significantly if create and initialize information was available in, for example, a static variable, for each component.

A utility process, which can be executed in batch mode, creates components in BC Prototyper format, from `.class` files, and puts them in a catalog. The catalog is a simple file structure that matches the package structure in the imported class files.

Once a component is in the BC Prototyper work area, it can be used as any class defined by the user. The contents of the server part should not be changed, but the view specification in the component can be changed.

Figure 12 The Greed game model (redrawn from BC Prototyper screen image)

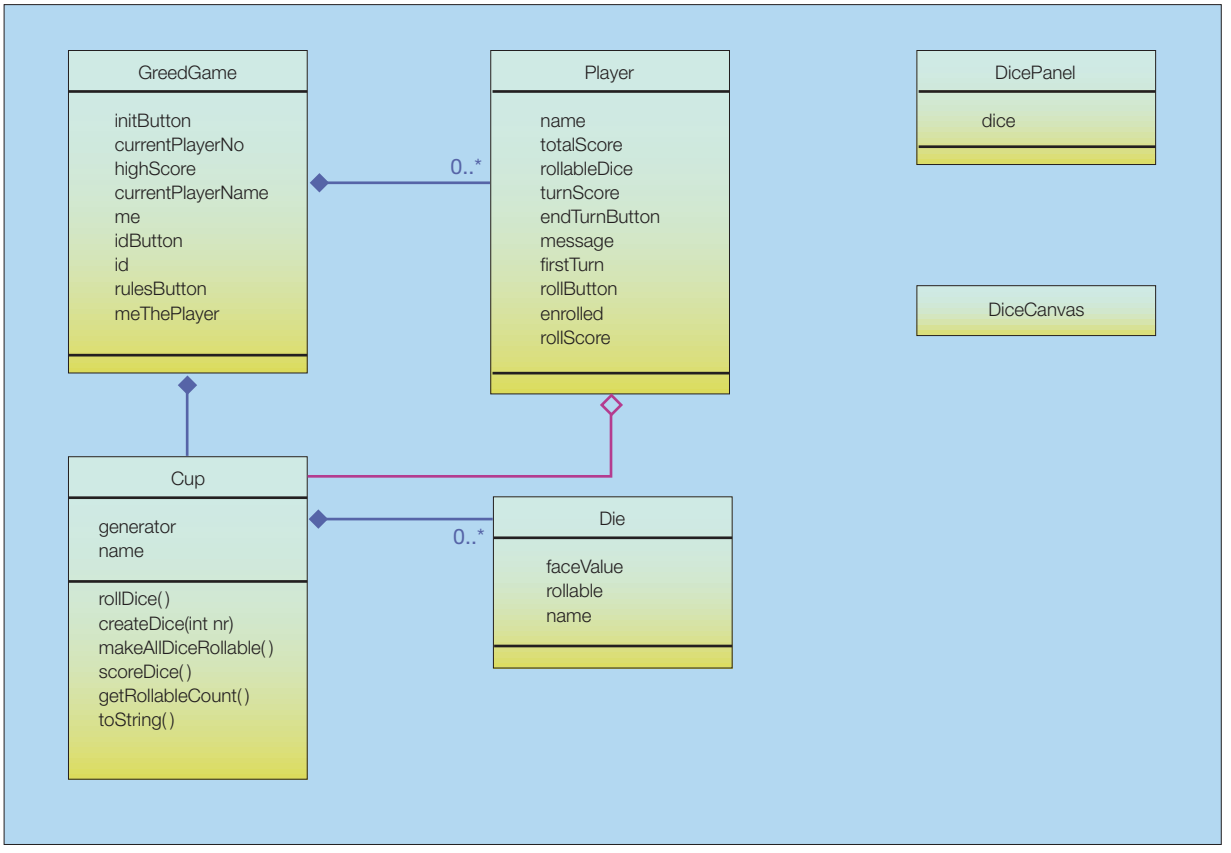


Figure 11 shows the result of running the code generated from the default view specification in the Address component, one of the SanFrancisco common business objects.

BC Prototyper will generate appropriate client code to access component objects from the server. It will also handle aggregations or other relationships established in the model.

Another example

As a last example, we would like to show an application that does more than define a few data objects and generate a maintenance interface. In his book *Object Lessons*,[8] Tom Love describes a simple dice game, called "Greed." The rules for the game are the following:

- Each player rolls five dice, in turn, until a player has scored more than 5000 points and all players have had an equal number of turns. (Each die has six faces; each face has a unique value from one to six.)
- The rules for scoring are:
  - Three of a kind—100 points times face value of one of the three dice
  - Three ones—1000 points
  - Single one—100 points
  - Single five—50 points
- Dice that did not add to the score during a turn can optionally be rerolled.
- If all dice scored during a roll, they can all be rerolled.
- If no die scores on a roll, the player is "bust" for that turn (the turn ends with no points).
- A player must score at least 300 points in the first turn to enter the game.

Love's book describes a competition for implementing the game, where solutions are programmed in

Table 2 Methods and responsibilities for Greed game classes

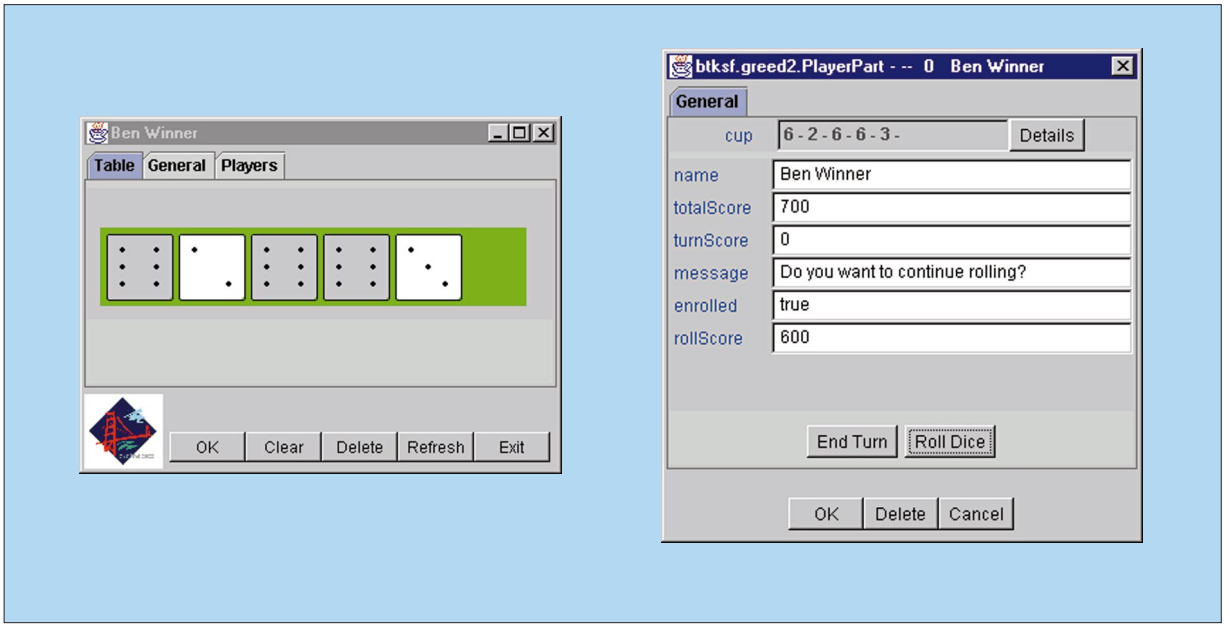
Class: GreedGame	
public void initialize( )	Create the dice. Ask all players to reset their scores. Select first player and ask the player to start.
public void selectNextPlayer( )	Find the next player to play. The player gets the cup, indicating that this is the current player. If at the end of a round, see whether to end the game.
public boolean endOfGame( )	From the list of players, select the first player holding the highest score, and see if it is a winning score. (This version of the game does not check to see if other players have the same high score.)
public void identifyPlayer( )	Provide the name of the current player.
public void businessObjectChanged( )	If invoked from the current player, start the player's turn.
Class: Player	
public void playTurn( )	Roll the dice that are rollable and compute the score. If continuation is possible, allow player to choose to continue.
public boolean turnCanContinue( )	Each player's first roll in the game must be at least 300 points. The new number of rollable dice must be smaller than the previous number of rollable dice.
public void endTurn( )	The end-turn button was pressed. Add any score to the total.
public void startTurn(CupPart c)	Start a turn for this player.
public void initialize( )	Initialize the player values for a new game.
public void endTurnClicked( )	Reset the message and end the turn.
Class: Cup	
public void rollDice( )	Roll the dice that are rollable, then make them not rollable.
public void createDice (int nr)	Create the dice in the cup.
public void makeAllDiceRollable( )	
public int scoreDice( )	Score the dice, then make the appropriate dice rollable.
public int getRollableCount( )	
Class: Die	
public boolean isRollable( )	

C++, Object Pascal, Smalltalk, Eiffel, etc. In the solutions presented in the book, players would share a single workstation. In our case, we would like each player to have his or her own workstation and play the game across a network. SanFrancisco, with its *notification* facilities, offers a good platform for this game.

The model in Figure 12 shows that the Greed game has a number of players, and exactly one cup. The cup has a number of dice (five in our case), and a player holds a cup during a turn. Table 2 shows a brief description of the methods and responsibilities for each class.

What is important to note with respect to the distributed version of the game we developed are the methods identifyPlayer( ) and businessObjectChanged( ). There is one persistent instance of GreedGame, located on the server. When something

Figure 13 Greed game as player “Ben Winner” sees it



changes the state of this object, all players are notified through the SanFrancisco notification service. The “businessObjectChanged” message is sent by a player when the player does something to change the state of this persistent object. To join the game, a remote client creates a Player object and adds it to the set of players in the GreedGame object. The client holds a local part of the Player object. The player uses “identifyPlayer” to obtain the name of the current player from the GreedGame object. By comparing that name to the player name in the client, the player knows when to play.

Note also that the persistent GreedGame instance makes it possible for players to stop the game temporarily, then come back the next day and continue.

The GUI for the game consists of generated code, although it is clear that some coding has to be done to show the dice graphically. Each player will be notified of updates in the game and will be able to see in real time what the values are of the thrown dice, and who is the current player (see Figure 13).

The Greed game example serves several purposes:

- It demonstrates SanFrancisco’s distribution and notification capabilities.
- It demonstrates the power of model-driven development.
- It shows customizability of the user interface of applications developed with BC Prototyper.

Unlike the address book example, the Greed game does not demonstrate the use of SanFrancisco common business objects.

Conclusion

This paper has described Business Component Prototyper, a tool that is itself a prototype demonstrating new possibilities for development tools for Java environments. It supports both modeling and coding activities in an integrated way.

The core part of the tool is developed using the tool itself. The use of code generation allows quick update of the meta-information used in the tool and its user interface.

All user actions, and model presentation functions, are encapsulated in bean-like modules. The functionality of BC Prototyper can be adapted to the user by configuration. Specific configurations for prototyping in SanFrancisco, stand-alone applications, tool developers, or SanFrancisco utilities can be made.

BC Prototyper has a flexible, easy-to-maintain, and customizable code generator that makes it possible to develop more powerful code generation more quickly.

BC Prototyper supports components, particularly SanFrancisco components, in a black-box fashion. This allows the tool to be kept small, with the ability to handle very large models, such as the SanFrancisco model.

Acknowledgments

First thanks are for Peter van Emde Boas and Maarten van Nouhuys, who helped and encouraged me during earlier stages of this project. For the development of Business Component Prototyper, I want to thank David Weilers, a student, who made the initial plug-in structure and the Java source code importer; Jeff Ryan, who was the project leader and contributed many of the SanFrancisco utility-related features; Charu Puri, who worked on the user interface and the code generation templates; Bob Schmidt, who managed it all; Julius Peter, who supported the ideas; and last but not least, Eddy Blum, who made it possible to spend more time on BC Prototyper than just evening hours.

Cited references and notes

1. B. J. Cox, *Object-Oriented Programming: An Evolutionary Approach*, Addison-Wesley Publishing Co., Reading, MA (1987).
2. D. A. Taylor, *Object-Oriented Information Systems: Planning and Implementation*, John Wiley & Sons, Inc., New York (1992).
3. E. Yourdon, *Object-Oriented Systems Design: An Integrated Approach*, Prentice Hall, Inc., Upper Saddle River, NJ (1994).
4. UML is the language that resulted from cooperation among well-known object technology methodologists: Grady Booch, James Rumbaugh, and Ivar Jacobson.
5. The publication of this paper does not imply that IBM will develop tools based on the BC Prototyper architecture. It also does not imply that BC Prototyper will be supported with new versions of SanFrancisco. The version of BC Prototyper described here contains experimental new functionality, such as component support, which is not available in the version provided with SanFrancisco v 1.40. BC Prototyper is intended for evaluation, education, and simple prototyping use. For the development of production applications, other modeling tools should be used.
6. The oldest (and very successful) multinational corporation in history was the VOC (United Company for the East Indies), a trading company that existed from 1602 to 1799. In 1618, the VOC sent the ship "Nieuw-Hoorn," with Willem IJsbrandz ("IJ" is the Dutch way to write "Y") Bontekoe as captain, on a voyage to Batavia (now Jakarta) on the island of *Java* to trade silver for spices, tea, and *coffee*. Near the end of the voyage the ship exploded—the brandy caught fire and lit the gunpowder. Captain Bontekoe was saved and led about 70 men, in a small boat with sails made of clothing and almost no food or water, navigating by the stars, safely ashore. He wrote a book about this trip, which is still, 350 years later, the most famous travel adventure book in Holland.
7. W3C, Extensible Markup Language (XML) 1.0, http://www.w3.org/TR/REC-xml/.
8. T. Love, *Object Lessons: Lessons Learned in Object-Oriented Development Projects*, SIGS Books, Inc., New York (1993).

Accepted for publication December 21, 1999.

Hendrica (Ghica) van Emde Boas-Lubsen *(electronic mail: emdeboas@cs.com).* Ms. van Emde Boas-Lubsen is an independent consultant, recently retired from IBM after 30 years with the company. During the last two years, she was a member of IBM's Europe, Middle East, and Africa (EMEA) SanFrancisco support team. She is the primary author of the Business Component Prototyper tool for SanFrancisco. Ms. van Emde Boas-Lubsen has an extensive background in object technology and relational database technology. She holds a degree in mathematics from the University of Amsterdam in the Netherlands.