Designing Procedural 4GL Applications through UML Modeling

Shiri Davidson, Mila Keren, Sara Porat, Gabi Zodik

IBM Haifa Research Lab
Matam - Advanced Technology Center
Haifa 31905, Israel
{shiri, keren, porat, zodik}@il.ibm.com

Abstract: This paper presents a Unified Modeling Language (UML) model for VisualAge Generator (VG) business-oriented applications. This model was defined to bridge between two different models, the object-oriented UML design model and the VG programming model, which use a procedural high-level 4GL. We introduce a prototype tool named VG UML Modeler which is implemented as a Rational Rose Language Add-in for designing VG applications. This tool provides developers with the ability to create a UML model for VG applications, check its compatibility, and generate the appropriate VG 4GL entities. The paper describes a methodology of the design process for a sample application.

1 Introduction

Today, the most common modeling language used worldwide is the Unified Modeling Language (UML). Its goal is to provide a common modeling language that supports good practices for designing Object Oriented (OO) software applications, such as encapsulation, separation of concerns, and reuse. UML defines different diagrams for describing the different aspects of an application. By using UML, a project's design process can be organized in a way that clients, analysts, programmers and others involved in the system's development can understand and agree on its design prior to the implementation phase. More and more companies are learning the benefits of designing their applications in UML. The ability to design applications with UML has becomes one of their main priorities in choosing a development environment.

Our work addresses a significantly different programming model, the IBM VisualAge Generator (VG). This tool is designed for the development of business applications, and is based on a procedural high-level fourth generation language, (4GL). This language seems very far from the OO approach. Fortunately, the power of UML is not limited to OO languages and it has even been applied to support the design of database schema. Our contribution is in proposing another application of UML design for a non-OO context. This is achieved by defining a UML model that incorporates the most valuable OO features. The proposed VG UML Model includes a set of entities, presented by UML classes with special VG-oriented stereotypes, and the UML relationships between them. Most of these entities are either based on VG 4GL parts or encapsulate them. Through the analysis of the VG application structure we found that encapsulating 4GL data entities and the related function entities into objects can be beneficial to the design process. We define Program objects to be based on the 4GL program part that holds the computation units of the VG application. In addition, we introduce other types of objects, such as the ExternalObject (based on 4GL SQL records) and the ViewObject (based on TUI Maps or Web UI records), which encapsulate data with the related functions required for access.

The proposed model strongly supports the 3-layered Model-View-Control (MVC) architecture paradigm. During the design process, the application is partitioned into the corresponding layered structure, where entities of each layer organized into groups of related entities representing different kinds of objects.

As a result of our research, we created the VG UML Modeler, which was developed as an Add-In for Rational Rose. It provides an actual link between UML and the VG 4GL. Being based on the VG UML Model, it supplies a distinctive design environment for designing VG 4GL applications. Using it, designers can create and modify model entities using wizards, check their model's correctness, and generate their model into a formatted file that can be imported into the VG environment. The tool's generation mechanism uses a map of UML classes to 4GL parts, based on the proposed VG UML Model. The created 4GL entities include program parts and data related entities (such as records) that are completely defined, and function skeletons that can be filled by the developer in the VG environment.

In our work, we focus on the static structure of the UML model (especially class diagrams) that is similar to other Rational Rose Add-Ins (such as Java, C++, etc.).

The VG UML Modeler is targeted for designers familiar with UML, who are either skilled 4GL developers or OO developers with some skills in 4GL. The modeler can also be used to attract OO developers that are not familiar with VG 4GL by enabling them to learn the basic principles of this language using a UML notation.

The following section describes the specifics of VG applications, their architecture, and development. Section 3 introduces the proposed UML model for VG 4GL applications. Section 4 describes the implemented VG UML Modeler, followed by Section 5 that demonstrates the proposed design process of a sample VG application using this tool. Finally, Section 6 summarizes our work, together with some possible directions for future work.

2 Business Applications Development using VisualAge Generator

2.1 Architecture of VG 4GL Applications

The IBM VisualAge Generator (VG) is a high-end system deployment environment for building and deploying e-business and multi-tier client/server applications. The purpose of such applications is to provide connectivity between end-user activities and back-end data repositories.

The development process of VG applications consists of creating numerous 4GL parts. There are basically two different kinds of 4GL parts:

- Logic parts which include program and function entities. These parts represent the application logic. Function entities are implemented using a special high-level 4GL logic scripting. A program entity is composed as an ordered list of function entities and describes a certain applicability or usage (e.g., a stand-alone main program such as TUI or Web, or one that is invoked by it using callable functions).

- Data elements. These specify structured data, referred to by various kinds entities, such as Record, Data Item, Map, Map Group, and Message Table. Record entities can be further categorized according to the context of the structured data (e.g., SQL Record for DB repositories, UI Record for Web application, Working Storage (WS) Record for data manipulation). A Record entity is composed of Data Items. Map and Map Group are additional user interface elements that are used in TUI applications.

A typical VG application consists of a set of 4GL programs. The control between these programs is transferred using special 4GL statements: CALL, XFE, or DXFR. In Web applications, a program can also be invoked via a special field in a UI Record named Program Link.

The created programs can be tested in the VG development environment directly against existing repositories. The final 4GL code can be generated into the real platform code according to specified generation parameters. The generated code should be installed together with the corresponding VG Servers according to the user's instructions.

The current version of VG is a plug-in to the VisualAge (VA) Java/Smalltalk environment, and allows the development of whole applications. For instance, the GUI part of an application can be composed using the built-in VA composition editor. The next version is planned for incorporation into the new family of IBM development tools with extended support for Web applications.

VG programs have a wide range of applicability. In this paper, we restrict ourselves to the context of business applications. Therefore, we concentrate on certain VG entities, while others such as records related to files, queues and other similar organizations are not considered.

2.2 Specifics of Business Applications

Business applications are mostly designed to connect end-user activities with different kinds of data repositories. Different architectures are described in the literature [RR00a] to bridge between these separate worlds, e.g. the 3-tiered Business architecture. In this kind of a solution a business application is divided into three tiers: the Relational Database tier, the User Interface tier, and in between these two lies the Business Object Model tier. Thus, the User Interface tier does not have any direct interaction with the Relational Database tier, but instead, it interacts with the Business Object Model tier that is responsible for the required communication. This separation allows us to develop each tier independently, and their common interface can be very compact.

The VG environment takes the Relational Database tier out of developers' concerns (this tier is supported automatically by generated code provided for VG Servers). For this reason, the architecture described above is not ideal for VG applications. Therefore, the proposed design structure is influenced more by the MVC model (Model/View/Controller), which is used by several graphical frameworks such as Smalltalk, etc. In the case of VG applications, the Model part corresponds to the

Business Object layer, which is responsible for interacting with both database repositories and user interfaces, as detailed below.

3 UML Model for VG 4GL Applications

Currently, there is no OO design methodology that can be directly applied to VG applications. The regular design process for such applications is procedural oriented, and is based on program flow. In addition, it should address the appropriate setting between client/server programs.

In the following subsections we apply the paradigm of Model-View-Control (MVC) layered architecture for VG application modeling. In addition, we will focus on two of the main aspects of OO approach: encapsulation and reuse of 4GL entities. The proposed model is defined so as to facilitate mapping of its entities to the VG 4GL application parts.

3.1 Applying Business Application Layered Architecture

The typical business application tiered architecture is not fully applicable for VG modeling because, as we mentioned above, the VG environment automatically supports connectivity to the Data Repository layer by generated code. For VG developers, the main modeling task is concentrated on the design of program units handling control activity between the user interfaces and business data structures.

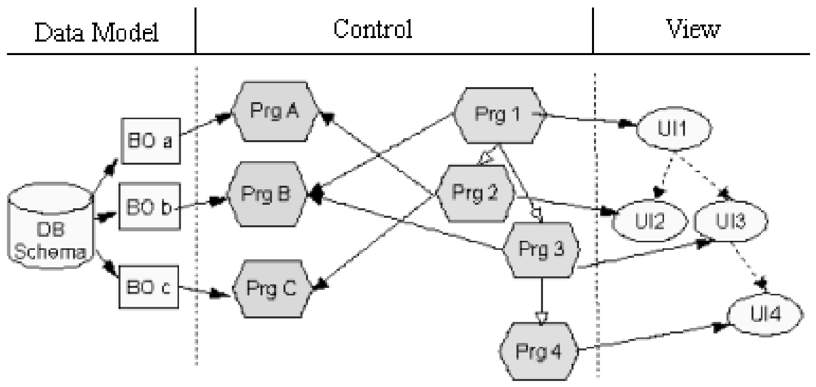


Fig. 1: MVC for VG Applications

It seems natural to apply the popular layered architecture of Model-View-Control (MVC) in the context of VG applications. A typical VG application is thus divided into the corresponding layers by having the program entities populate the Control layer. Various types of Data elements and their related functions are distributed between the View and Model layers. The resulting architecture appears as follows (see Fig. 1):

- Model layer – holds external SQL data with their related functions.
- View layer – holds UI Records with and their related functions.

- Control layer– holds the programs with data and their corresponding functions.

We observe that among these three layers, the Model and View layers are the less flexible ones. Their design is strongly coupled with external specification and data. In contrast, the Control layer leaves much room for designers to use various approaches. For example, some alternative architectural design models are:

- A single program for the entire application.
- Several programs, each handling a specific external data (SQL Record or UI Record).
- A cluster of programs that address the same external data.
- Ad hoc programs without consistent relation to any data

The goal of a proper design process is to guide the developer in building the design scheme appropriate to the specifics of the application, as demonstrated in Fig. 1.

3.2 Encapsulating 4GL Entities into Objects

The proposed model consists of two entity categories: core entities that reflect existing 4GL parts and compound object-like entities that have no direct analog in 4GL. Creating these compound objects was a natural observation of the design once we proposed to encapsulate data entities with related function entities. This approach directs designers to adopt OO encapsulation by gathering the functionality and data into independent objects.

The VG UML Model includes a set of VG UML entities, presented by UML classes with special VG-oriented stereotypes. Fig. 2 presents the corresponding classes and the relationships between them. We start with the main procedural part of the application, as depicted by the VG 4GL program entity. This entity serves as a container for both data record sets and functions that use these sets. Thus, the corresponding encapsulating object in the control layer is defined accordingly, as the Program object. One of the inherent characteristics of this object is that it has a single public method from which its ordered set of private functions is invoked.

Next we introduce encapsulated objects in the other layers, namely the View and Model layers. This allows us to spread the complexity of the application logic into smaller units, thereby reducing the complexity of the Program object.

The introduced VGObjects encapsulate data information via VGRecords and their bounded functions. We define three additional types of objects:

- *ExternalObjects* that encapsulate SQLRecords that represent external storage (corresponding to the 4GL SQL records) together with functions related to these records (e.g., INQUIRE, ADD, UPDATE functions)
- *ViewObjects* that similarly encapsulate UIRecords (corresponding to the 4GL's User Interface records) with their related functions (e.g., CONVERSE, DISPLAY functions)
- *GeneralPurposeObjects* that encapsulate WSRecords (naturally corresponding to the 4GL Working Storage records) and the corresponding execution functions that work on them. These objects are shareable and thus can be used, for example, as error handling units.

As mentioned above, the core objects are based directly on existing 4GL entities (data items, records, tables, etc.). These objects are used to build the compound object and allow the designer to express low-level design issues with a notation similar to the 4GL logic, as specified above. The difference is the manner in which their internal structure is specified. A 4GL record does not contain subrecords but a hierarchical 4GL data item structure. This prevents the reuse of identical substructures between different 4GL records. In contrast, the VGRecord is composed of either DataItems (with no nested structure) or other VGRecords as subrecords (containing nested structure). This provides reuse of the subrecord's structure in two possible ways: by including previously created entities as a new record's class attributes and by defining association relations between them.

3.3 Defining Relations Between VG UML Entities

One of the important added values of the OO approach is the possibility of reusing previously created elements. This can be achieved by declaring the different UML relations between objects. In this section, we define relations between the model entities. Some refer to interrelated objects, while others connect between different types. The relations address inheritance relationships, composition, and other associations between objects.

As regularly defined, inheritance relationships allow the subclass to reuse the parent's features and possibly refine them. This kind of relationship is introduced for the newly defined Program entity and VGObjects, thus providing these entities with the ability to inherit attributed records, associated records, and methods. Methods can further be overridden.

Special features are specified for the inheritance between Programs. As described above, a Program has a single public method responsible for invoking a list of internal functions. A subclass Program can override this public method by modifying the corresponding list of functions.

Inheritance between ExternalObjects can reflect the concepts of mapping objects to tables ([RR00a], [RR00b], [KJA93]). A subclass may refine the attributed records of its parent in two ways: either by extending their column pattern (distributed inheritance), or by extracting certain subsets (collapsed inheritance).

Composition relationships can be applied between VGRecords. A VGRecord can be constructed by associating certain DataItems, or other VGRecords. The associated entities become nested subparts of the source composed record. This allows yet another level of reuse among related entities.

Associations are further defined between entities of different stereotypes, depending on the kind of associated entity. Some possible associations and their corresponding constraints are:

- Primitive entities, those that are not composed of other entities, (e.g., DataItems) cannot be associated as a source with other entities.
- VGRecords can only be associated to DataItems or to other UIRecords:

- ViewObjects and ExternalObjects can only be associated with the corresponding kind of VGRecords (UIRecord or SQLRecord, respectively).
- Programs can aggregate other VGObjects and WSRecords. In addition, they can be associated with other Programs using special stereotyped relations (CALL, XFER etc.).

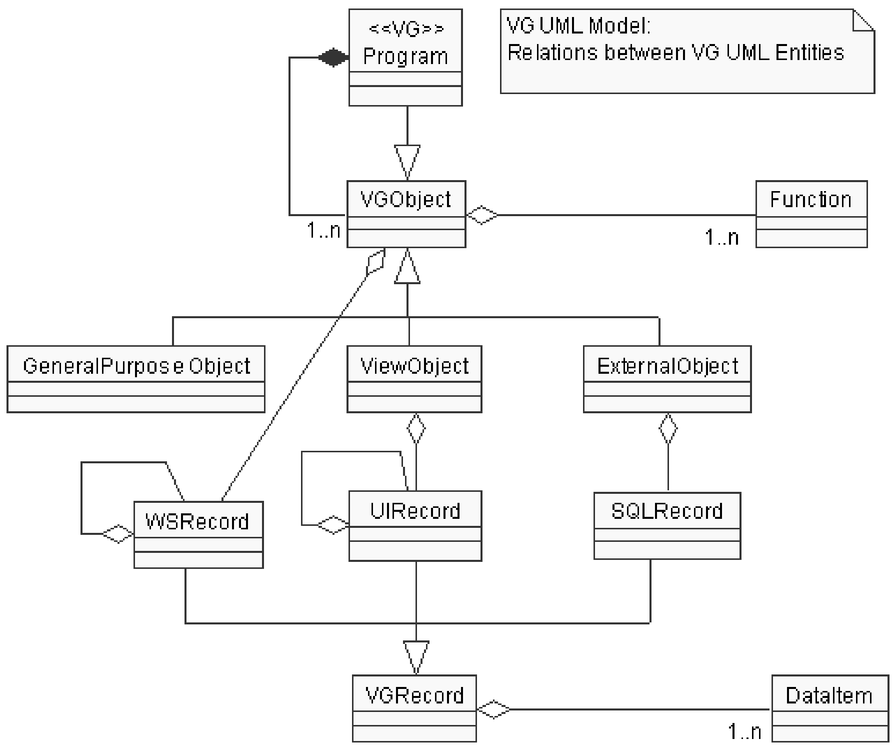


Fig. 2: VG UML Model Entities and Their Relationship

The model thus incorporates widely used UML relationships, but certain constraints were introduced in order to enable reasonable mapping to 4GL entities. As discussed below, one of the major features of the tool that supports this VG UML model is demonstrated by its checking capability.

4 Overview of the VG UML Modeler

The VG UML Modeler is implemented as a language Add-In to the Rational Rose tool. It was developed to provide VG 4GL developers with a special environment for designing their VG applications through UML. The tool is based on the model described in the previous section. The tool's environment enables developers to design VG 4GL applications using our newly introduced entities, check compatibility of the designed model to the VG UML Model, and generate the designed UML entities into VG 4GL entities.

The VG UML Modeler provides different features including the VG Framework, class creation and modification wizards, checking, and generation support. To maintain this, the VG UML Modeler registers its unique specifications in the Rose registry. Among these specifications are files containing definitions of the UML extensions needed to support the VG UML Model and a list of menu items added to Rose's main Tools menu list. This section describes these various issues in more detail.

4.1 UML Extensions for the VG UML Modeler

Standard UML provides several extension mechanisms that enable UML to be adapted to different types of systems, domains, methods, and processes. The VG UML Modeler uses the stereotypes and properties UML extension mechanisms to embrace UML for VG 4GL applications. These extensions are defined based on the VG UML Model's special characteristics and stored in several files registered with the VG UML Modeler in Rational Rose.

A stereotype is defined as "a variation of an existing model element with the same form (such as attributes and relationships) but with different intent" [RJB99]. The VG UML Modeler defines new stereotypes for UML classes, associations, and dependency relationships. These stereotypes represent our VG UML Model's newly defined entities and special relationships. The following includes the list of the VG UML Modeler stereotypes:

- Class stereotypes: ExternalObject, ViewObject, GeneralPurposeObject, Program, SQLRecord, UIRecord, WSRecord, DataItem
- Association stereotypes: Xfer, Dxfr, Call, Program Link
- Additional dependency stereotypes (Bind , Usage, Primary, Based On)

Another UML extension mechanism used by the VG UML Modeler is the properties mechanism. A property is "a generic term denoting a named value conveying information about the model element" [RJB99]. The VG UML Modeler defines new properties for UML classes. These properties are used to enhance classes with additional characteristics. They are needed for specifying VG related values that are not available through ordinary class definitions (methods and attributes). One of them is the "Ordered List" class property that is used for classes when some of their characteristics have significant order. For example, Program classes use this property to store the ordered list of main methods intended to be executed according to a specified order.

4.2 VG Framework

In the Rational Rose tool, a framework is "a model with predefined model elements that is used to model a certain kind of system." [RR98]. When creating a new model from a framework, it is initialized with the framework's predefined architecture.

Through our research, we've seen that dividing typical VG applications design into Model, View, and Control related classes (packaged accordingly) simplifies the design .

Therefore, we’ve created the VG Framework that initializes models with these packages and includes an additional corresponding three-layered diagram designated for designing connectivity between the different applications parts. VG 4GL designers can choose either to design their applications using the VG framework or to ignore it and start their application’s design from scratch.

4.3 Creation and Modification Wizards

The VG UML Modeler contains special wizards for creating and modifying related classes. These wizards assist designers in setting and updating class specifications using class stereotype-oriented specification views that differ for each of the different stereotypes, depending on its basic characteristics. Both the creation and modification wizards use these specification views.

The creation wizard assists users in creating new classes. It consists of three states that are traversed via the wizard’s "Back" and "Next" push buttons. In the first state, the designer selects the class’s VG-oriented stereotype, then specifies its name and package, and finally uses the specification views for setting the class specifications.

The modification wizard opens directly the class’s specification view according to the class stereotype. Using the wizard's 'OK' and 'Apply' buttons, designers can save the modified class specifications.

4.4 Checking Support

The VG UML Modeler’s checking support is responsible for verifying that the designed models are compatible with the proposed VG UML Model.

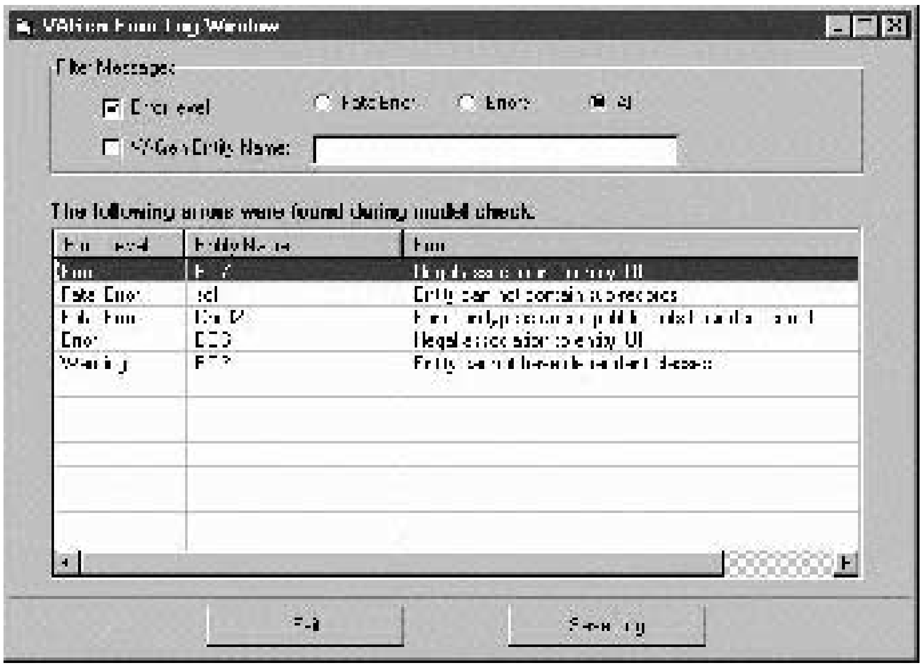
Each class in the model is represented by special characteristics. These characteristics are gathered from the class attributes, methods, outgoing relationships to other classes or data model tables, and the class’s UML property values. In order to check the designed model, we should go through all classes and their characteristics and inspect their consistency with the specified class stereotype. For example, the ViewObject stereotyped class can have outgoing associations to UIRecord and WSRecord stereotyped classes and other VGObjects. However, an association to an SQLRecord stereotyped class is illegal; whereas for ExternalObject stereotyped classes this relationship is valid. The VG UML Modeler checking support verifies VG UML Model classes as follows:

(1) Creates special class stereotype handler for each class stereotype of the *VG UML Model*. These handlers implement a common interface that contains methods for checking a class entity according to its special stereotype related characteristics.

(2) Passes each queried stereotyped class (that has not been checked already) to its equivalent handler. Using the handler, the class is queried for its correctness. Through this process, the handler is queried for other classes that need to be checked as well as to confirm the classes correctness. To conclude correctness, these classes, along with the inquired class, are to be checked.

(3) Displays any errors found to the user using an Error Summary view. This view helps designers filter the errors in order to simplify their understanding. The errors found are categorized into three levels according to their influence on the process of the

model generation to the VG-format files : fatal, for classes that cannot be generated; regular, for classes that can be partially generated (only their non-erred parts); and warning, for cases when we suspect the designer intended to do something else, but the class can still be generated. Fig. 3 includes an example of an Error Summary view. For more information on the generation support, see section 4.5.


VAlion Error Log Window

Filter Message:

[✓] Error Level ( ) Fatal Error ( ) Error [● All]

[ ] VGGen Entity Name:

The following errors were found during model check:

Error Level	Entity Name	Error
Error	[illegible]	[illegible] to entity UI
Fatal Error	sd	Entity can not contain [illegible]
Fatal Error	[illegible]	[illegible]
Error	DDD	Illegal association to entity UI
Warning	[illegible]	Entity [illegible] dependent classes

Fig.3: Error Summary View Example

In order to be able to inform designers about possible errors in their models during development, the checking support checks the models classes in three different scopes:

- Internal Checking –checks internal class parts that can be modified only through the creation and modification wizards. This checking is invoked on saving a newly created or modified entity through the wizard's life cycle.
- Standalone Checking – checks class characteristics that can be modified in the Rose environment without opening the modification wizards. This checking is invoked prior to the display opening of the class modification wizard.
- Broad Checking – activates standalone checking on the class; additionally, it activates broad checking on all classes related to it. In particular, it activates broad checking on the class’s super class, nested classes, and aggregated/composed/associated classes. This checking ensures that the class and all its dependent classes are error free. It is invoked on a explicit checking request and prior to a generation.

4.5 Generation Support

The generation support is responsible for generating the designers' VG UML Models to a special ESF format file, which is recognized by the VG environment.

Prior to the model's generation, all the models classes and relationships are checked (using a broad checking mechanism described in the previous section). If errors are found, they are displayed to the user. The user can decide whether or not to continue the generation in spite of the errors, although, erred entities are not guaranteed to be generated. Following checking, each non-erred entity is passed for generation.

The generation mechanism is similar to the checking mechanism. It uses the same VG class stereotype handlers, which in addition to implementing the checking interface, implement a common generation interface. This allows each handler to be responsible for supplying the class generation result according to the class's special characteristics denoted by its VG class stereotype. The VG UML Modeler requests the generation result from these handlers. The result is then piped to the user's target ESF file.

The following section describes the generation result of the different stereotyped classes.

4.5.1 Generating VGObjects

The VGObject stereotyped class generation results in the generation of the class super class (if exists), the class itself, and all its associated VGRecord stereotyped classes. Throughout this generation, different 4GL entities are created.

The standalone VGObject stereotyped class generation results in the following 4GL entities:

- Class operations are generated as 4GL functions. To preserve the class's OO encapsulation in the non OO 4GL, we use a special naming convention for the generated 4GL functions. The generated 4GL function name is a concatenation of the class name followed by the operation name (i.e., className_operationName).
- Class attributes are generated as VGRecords that are further transformed to 4GL entities.

4.5.2 Generating Programs

The generation of a Program stereotyped class is almost identical to the generation of a VGObject stereotyped class (described above) since Programs are subclasses of VGObjects (for more information see section 3). The differences in generating a Program as opposed to generating a VGObject are:

(1) A 4GL program of the specified type is generated

(2) As *a VGObject* stereotyped class, its operations are generated as standalone 4GL function (denoted to the class by the naming convention). Additionally, the generation results in a 4GL program that contains these generated 4GL functions according to the order the designer specified.

(3) All *VGRecords* that are associated directly or indirectly (from associated *VGObject* stereotyped classes) are generated as well and they are all specified in the 4GL program's Additional Record set.

5 Design Process of a Typical 4GL Application

The proper integration of the design and implementation phases shape a successful development process that results in high-quality software. To provide VG designers with a suitable environment that support the whole process of designing their VG applications, we have implemented the VG UML Modeler (as described in the previous chapter). A typical development process cycle includes the following steps:

(1) Create the application's UML model according to the functional specifications.

(2) Check the model and update as needed.

(3) Generate the corrected model.

In this section, we describe the design process of a sample 4GL application using our VG UML Modeler, which includes: recommended base application division provided by the VG Framework, creating classes, checking and generating them.

The application being designed is a simple banking application. You must enter your card number and social security number and push the Submit button. Following this, the banking system checks the entered information. If the information is correct, you are presented with your user account information.

5.1 High Level Application Partitioning and Design

The application design usually starts by analyzing the functional specifications of the application (concluded in creation of use cases). The following design process has an iterative nature that evolves from the high level abstractions to the lower level models and to the implementation specification details.

When defining an application's high-level design, the designer defines the application's building blocks without taking into account the VG 4GL characteristics. We have concluded that if a designer designs high level application classes while taking MVC application partitioning into consideration, the designer can considerably ease the low level application design because the designer will be able to reuse the high level design for the next design stage. The packages should contain the following classes:

- Model – contain all classes that handle database related activities.
- Control – contain all classes that are responsible for controlling the application.
- View – contain all view related classes.

To aid VG designers in partitioning their application accordingly, we have defined the VG Rational Rose Framework. By creating a new VG application that uses this framework, the new model will include these three base packages along with a corresponding three-layered diagram.

We start the sample application design by creating a new model using the VG Framework and specifying the model's default language to be VG. Now, we can design the high level building blocks of the application using regular classes. Each building block should be categorized as either a Model, View, or Control related class and is packaged equivalently. These high-level design classes will later be transformed into the base low level VG stereotyped classes.

5.2 Data Model

Prior to defining the applications data structures, we need to import the database tables that they are based on into the application's UML Model. The Rational Rose Data Modeler Add-In provides users with the ability to create/import database schema. Additionally, it provides a special diagram that aids the visualization of the table's structure, primary and foreign keys, and existing relationships between different tables.

The Data Modeler's tables are used in the overall design process of the VG UML application for defining data structures (e.g., SQL Record stereotyped classes) that are used to access these tables. These data structures are built depending on these tables. Thus, designers can define the data structures internal structure based on the tables columns .

One of the tables used in the sample banking application is named Card_Customer, which includes the Bank's distributed card numbers and the social security number of their recipients. This table is used in the application design when defining the data structure that will access it.

5.3 Low Level Application Design

After we've defined the application's high-level design and specified what tables it will be using, we need to lower the application's design and give it a VG UML notation. The first step is to stereotype the application's high level design classes with corresponding VG UML Model stereotypes. This is done using the following mapping:

- Control related classes are stereotyped as Program (VG Processing Nodes) classes.
- View related classes are stereotyped as ViewObject classes.
- Model related classes are stereotyped as ExternalObject classes.

Fig. 4 includes a caption of the wizard activated by double-clicking on a non-stereotyped model class.

This wizard is activated in order to aid the designer in selecting the corresponding class stereotype. Following this stage, the application's model contains different VG-stereotyped classes including the relationships between them (see Fig.5).

The next step is to define the application's data structures and functionality. The sample banking application design includes, among other parts, a Card ExternalObject that is responsible for accessing the Card_Customer database table. This class needs to include a CardInfo SQLRecord as its data and basic Create, Read, Update, and Delete (CRUD) operations on the database table.

To define the CardInfo class, the designer creates a new SQLRecord stereotyped class and using the dependency relationship, specifies that it depends on the Card_Customer database table (as can be seen in Fig. 7). Then he defines the record's internal structure. This is done using the tool's record wizard. He first indicates on which of the table's columns CardInfo's internal structure is to be based on. Then he specifies that this record is to be used by the Card class, by either associating the Card class to the CardInfo class or by adding the CardInfo class as Card's attribute.

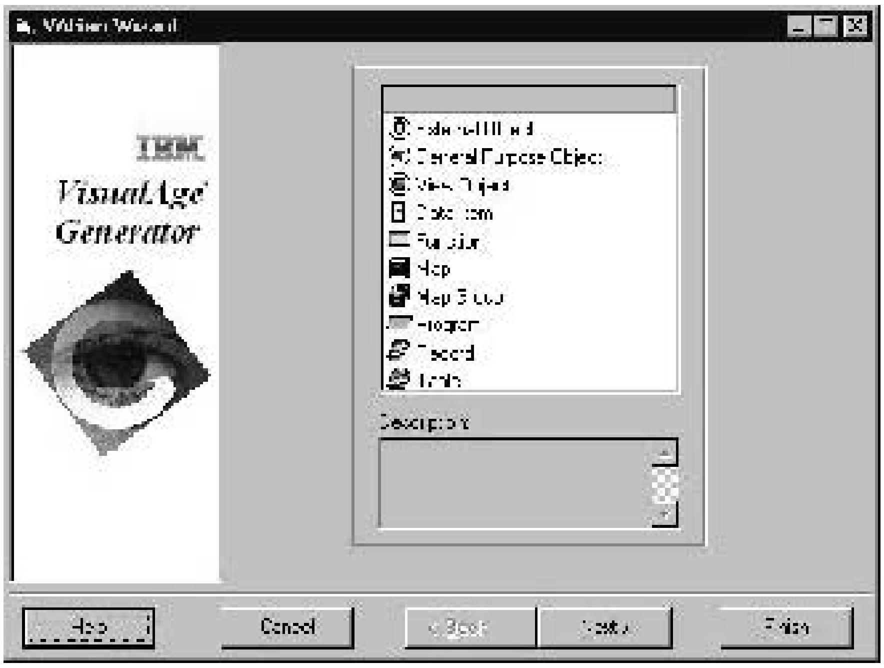


Fig.4: Selecting a Class Stereotype

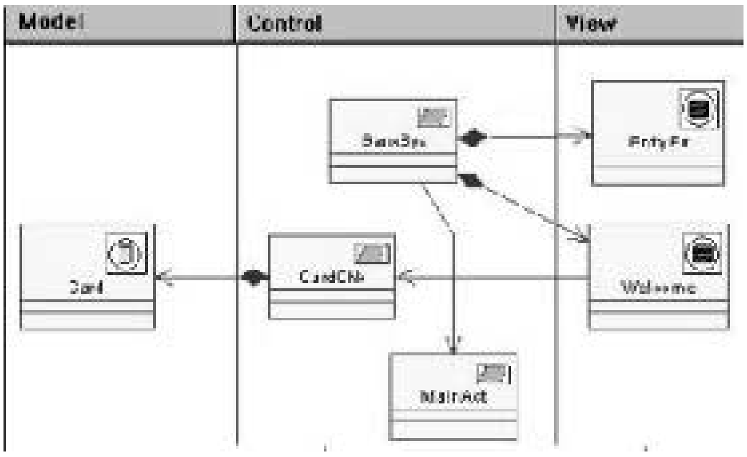
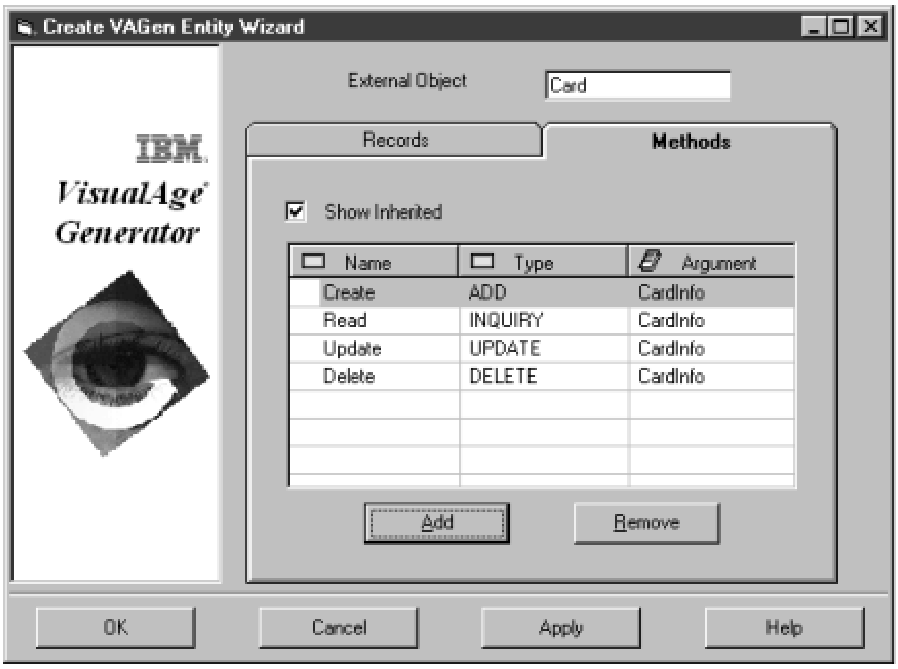


Fig.5: First Stage of the Banking Application Design

Once CardInfo is defined, the designer can specify Card's functionality using the tool's ExternalObject wizard. He adds to the Card class new CRUD methods that use the CardInfo record as their argument. Fig. 6 contains a caption of the ExternalObject wizard that displays Card's CRUD methods. Each method entry includes its name, type (one of the supported VG special purpose function types), and the CardInfo record as its argument.



External Object: Card

Name	Type	Argument
Create	ADD	CardInfo
Read	INQUIRY	CardInfo
Update	UPDATE	CardInfo
Delete	DELETE	CardInfo

Fig. 6: Card ExternalObject Methods

Fig. 7 contains a caption of part of the sample banking application design .The Model part contains the entities just defined to access Card_Customer table. The View part contains the application's Welcome and EntryErr ViewObjects responsible for displaying web pages through which the user logins to the application. Finally, the Control part contains the application's Programs. Among them are the CardChk Program that is responsible for checking the user's authentication, BankSys Program that is responsible for displaying the Welcome page and processing the user input, and MainAction Program that is responsible for continuing the interaction with the user following a successful login.

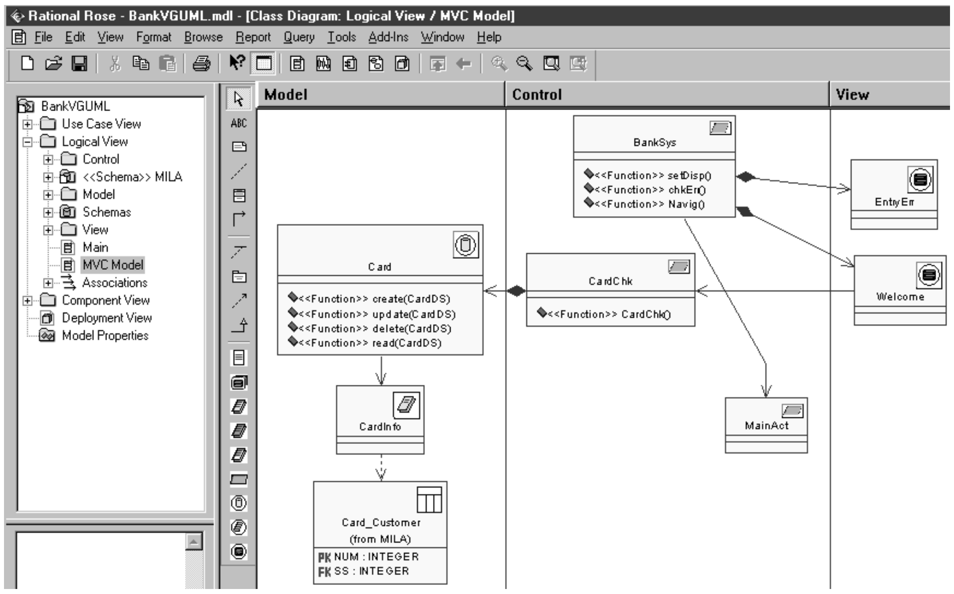


Fig. 7: Sample Banking Application Design

5.4 Checking and Generating the Designed Model

Once the designer has completed the VG application design with UML, the designer can complete the application's implementation in the VG environment.

To transform the designed model into 4GL entities that can be edited in the VG environment, the designer activates the generation support feature. This feature first verifies that all of the application design is compatible with the VG UML Model (using the checking support described in chapter 4). If so, the designed classes are generated into 4GL entities stored in a target ESF formatted. Then the developer can import this file into the VG environment for its complete implementation.

6 Summary

In this paper we describe the VG UML Model; a model for mapping between two different models, the object-oriented UML design model and the model behind the development of procedural VG 4GL applications. In addition, we introduce a prototype tool named VG UML Modeler that is implemented as a Rational Rose Language Add-in for the design of VG applications in UML.

The model and its realization via the *VG UML Modeler* have the following benefits:

- Bridge between a UML Design tool and the VG environment – provides a very productive way to design e-business applications. It does so by setting a special environment for designing VG applications in the popular Rational Rose design tool.

This includes special wizards for creating and modifying VG-oriented entities and others.

- Connectivity to Rational Rose Data Modeler Add-In – provides an easy way to create/import Relational Database schema using the Rose built in Data Modeler Add-In and reuse of its Relational Table entities as a basis for VG SQL records.
- Checking Support – provides an important mechanism for checking the design model, the various entities in the model, and the relations between them, making sure they can be mapped correctly to 4GL.
- VG Generation Support – provides the ability to transform UML entities into VG application parts. The outcome of the generation phase is an ESF file that can be imported directly into a VG environment, which results in the creation of the VG application's entities including data structures, programs, and functions templates.
- Further implementation is required, but the generated ESF file is aimed to significantly ease this process.

Future activities will focus on enhancing the current prototype tool by providing additional features and improving its usability . We are also investigating ways to express more abstract model concepts, which can provide better connectivity between the standard OO design and VG 4GL development.

References

[FPR00] Fontoura, M.; Pree, W.; Rumpe, B.; UML-F: A Modeling Language for Object-Oriented Frameworks: ECOOP 2000; S. 63-82.

[KJA93] Keller, A.M.; Jensen, R.; Agarwal, S.; Persistence Software: Bridging Object-Oriented Programming and Relational Databases: SIGMOD, May, 1993.

[LH96] Lindsey, A.H.; Hoffman, P.R.; Bridging Traditional and Object Technologies: Creating Transitional Applications. IBM System Journal, Vol 36, No. 1, July, 1996.

[RJB99] Rumbaugh, J.; Jacobson, I.; Booch, G.; The Unified Modeling Language Reference Manual: Addison-Wesley, 1999.

[RR98] Rational Rose 98 Using Rose: Rational Software Corporation, 1998.

[RR00a] Integrating Object and Relational Technologies: Rational Rose Whitepapers, March, 2000.

[RR00b] Mapping Object to Data Models with the UML: Rational Rose Whitepapers, March, 2000.