COMPUTING INTEGRITY
INCORPORATED
60 Belvedere Avenue
Point Richmond, CA 94801-4023
510.233.5400 Sales
510-233.5444 Support
510.233.5446 Facsimile

PROGRESS Application Partner

UML Profile for ABL

Version 0.4
12 November 2007
Thomas Mercer-Hursh

Introduction

A UML profile[1] is intended to provide a mapping from a particular domain language or the language of a particular methodology onto underlying UML constructs. This mapping is a combination of "stereotypes", which are terms from the domain or methodology that are equated to particular UML constructs, along with additional constraints, rules of "well-formedness", and identification of which particular elements will be used to model the elements associated with the domain or methodology. A typical example is the relationship between the stereotype «table»[2] in a relational database and a UML Class. This is not to say that a table *is* a class, but that Class makes the best base for modeling the characteristics of a table. The columns of a table are mapped to attributes in the class; validation rules on the columns are mapped to constraints; and triggers are mapped to operations.

One doesn't normally define a UML Profile for a programming language. Ideally, UML models are initially created as Computationally Independent Models (CIM) which are completely separated, not only from the programming language of the implementation, but also from any specifics of the architecture. These CIMs are then evolved manually or through Model-Driven Architecture (MDA) transformations to create Platform-Independent Models (PIMs) and eventually Platform-Specific Models (PSMs). It is only at the later stages of this process that the model acquires the form of any one specific programming language. Since the target language from most UML modeling is typically an OO language like C++, C#, or Java, there tends to be a fairly straightforward correspondence between the model and the corresponding language expression.

However, the motivation behind the current work is the desire to import existing Progress Advanced Business Logic (ABL) code into UML for the purpose of analysis and transformation and possibly eventually for generation of a revised system. This task presents certain challenges because the typical legacy ABL system is not written using OO constructs and UML is inherently biased toward OO in orientation. Therefore, it has been decided that it is appropriate to define a UML Profile in order to map the constructs found in ABL onto underlying UML elements. This will provide us with ABL-appropriate vocabulary on top of the UML tools. As a part of the analysis and transformation process we might later abstract the model away from these ABL-specific stereotypes, but the initial model constructed will be based on terminology which will clearly and unambiguously connect back to the underlying ABL code.

To provide this clarity and connection, we will define more stereotypes than might seem to be absolutely necessary and it is possible that later revisions will simplify this vocabulary, but it is felt at this time that the clarity and connection provided by a richer vocabulary in which each term has its own ABL-appropriate characteristics will provide a superior basis for analysis than would a more

[1] See http://www.omg.org/technology/documents/profile_catalog.htm for examples.
[2] UML Stereotypes are identified by being enclosed in double angle quotation marks or guillemets.

minimal set. Since this profile is very language specific, we will prefix each stereotype with "oe" for OpenEdge to make it clear that the characteristics of that stereotype are specific to the Progress database and ABL language. E.g., «oeTable» will clearly relate to the specifics of a table in a Progress database and may or may not relate to any other stereotype of «Table» which applies to any other database or which might be database independent.

This initial profile will describe the vocabulary and the mapping onto the target UML. Specification of the details of an XML Metadata Interchange (XMI) implementation of this profile will be deferred to a subsequent document. It is anticipated that initial implementation of tools using this Profile will write directly into the UML tool repository[3].

This profile consists of five sections. The first of these sections will relate to modeling the OpenEdge[4] database. The intent in this section will be to provide a complete specification such that sufficient information can be extracted from an existing Progress database to analyze and modify the model and then export a Progress .df such that a database conforming to the modified model can be constructed. The second section will deal with modeling the components of the ABL code. The third section will deal with modeling connections between that code and the database. The fourth section will deal with modeling logical or structural groupings of functional collections of code and tables. The fifth section will deal with the visual aspects of the user interface.

Profile for OpenEdge Database

The current OpenEdge metaschema includes many tables to support various aspects of dynamic management of the application in addition to defining the schema used by the application. For the purpose of this Profile, the initial target will be limited only to that information necessary to define the schema used by the application, i.e., that portion which is covered by the information in the corresponding .df file which defines the schema and the .st file which defines the physical layout of the database. There is also a significant amount of metaschema which exists to support Dataservers. This Dataserver portion will be neglected in the initial Profile and will be added later as needed.

A complete description of a OpenEdge database includes both the logical structure of tables and columns and the physical structure of areas. While only the former is required for analysis, the latter is required for creating a new database following modifications. The primary logical structure consists of one or more «oeTable»s, each containing one or more «oeColumn»s, collected into one or more «oeDatabase»s. Associated with these are «oeTableTrigger»s providing constraints and behavior on the «oeTable»s and «oeAssignTrigger»s providing constraints and behavior on the «oeColumns». Supplementing these are «oeSequence»s which provide sequential counters for use in the application. Index information is reflected in foreign and primary key designations on the «oeTables»[5]. Physical layout information is reflected in an «oeArea» definition to which «oeTable»s are assigned. Codepage and collation table information will be neglected in the initial release.

[3] Initial implementation will be targeted on Enterprise Architect from Sparx Systems (http://www.sparxsystems.com) because it is a tool which is very capable, popular in ABL community, and modestly priced. Other implementations will depend on market need.

[4] OpenEdge is the brand name for the product line related to ABL sold by Progress Software Corporation.

[5] Some databases provide formal join specification within the database, but this is not presently the case for an OpenEdge database. There are techniques by which joins can be imputed, but their accuracy and effectiveness varies by the naming standards used in the specific database. Some sites are likely to have already constructed formal specifications external to the database. As a part of this Profile we are providing structures for identifying actual table join relationships as manifest in the code so the specification that is a part of the database portion of this Profile should be considered the "theoretical" connections while that related to the code is the "empirical" connection.

«oeDatabase»

An «oeDatabase» is a particular collection of «oeTable»s. It maps to a UML Package and its properties are:

- oeDatabaseGUID – a unique identifier for the instance. Used for linking.
- oeDatabaseName – the name of the database from `ldbname()`. Maps to the Name of the Package.

«oeTable»

An «oeTable» corresponds to a database table and maps to a UML Class whose properties are:

- oeTableGUID – a unique identifier for the instance. Used in linking.
- oeDatabaseGUID – a unique identifier linking to the «oeDatabase» of which the table is a part.
- TableCanCreate – a list of users authorized to create new records in the table. Maps to a UML Tagged Value. From _File._Can-Create.
- TableCanDelete – a list of users authorized to delete records in the table. Maps to a UML Tagged Value. From _File._Can-Delete.
- TableCanDump – a list of users authorized to dump records from the table. Maps to a UML Tagged Value. From _File._Can-Dump.
- TableCanLoad – a list of users authorized to load new records in the table. Maps to a UML Tagged Value. From _File._Can-Load.
- TableCanRead – a list of users authorized to read records in the table. Maps to a UML Tagged Value. From _File._Can-Read.
- TableCanWrite – a list of users authorized to write new records in the table. Maps to a UML Tagged Value. From _File._Can-Write.
- TableCreator – id of the table creator. Maps to a UML Tagged Value. From _File._Creator. Will normally be "PUB" for regular tables and "SYSPROGRESS" for metaschema tables.
- TableOrigin – Value indicating whether table originated from ABL or SQL. Maps to a UML Tagged Value. From _File._DB-lang.
- Description – a description of the table. Maps to a UML Comment attached to the «oeTable». From _File._Desc.
- DefaultPrimaryKey – a flag indicating whether or not the primary key for the «oeTable» is still the default. Maps to a UML Tagged Value. From _File._dft-pk.
- DumpName – the name used when exporting the table to an external file. Maps to a UML Tagged Value. From _File._Dump-name.
- TableLabel – a name for the table used in displays. Maps to a UML Tagged Value. From _File._File-Label.
- TableLabelStringAttr – string attributes[6] for the TableLabel. Maps to a UML Tagged Value. From _File._File-Label-SA.
- Name – name for the table used in programming. Maps to the name of the UML class. From _File._File-Name.
- Frozen – a flag indicating whether or not the table is frozen from changes. Maps to a UML Tagged Value. From _File._Frozen.
- Hidden – a flag indicating whether or not the table is hidden in ordinary displays of tables available. Maps to a UML Tagged Value. From _File._Hidden.
- PrimaryIndexGUID – indicates the primary index for this table; from _File._Prime-index. Maps to a UML Tagged Value.

[6] String attributes are T, R, L, C, U, and number of characters. These control the amount of space allocated in r-code for the string and various properties in handling the string. See OpenEdge Development: Basic Database Tools.

- Owner – a field indicating the id of the table owner. Maps to a UML Tagged Value. From _File._Owner. Will normally be "PUB" for regular tables and "SYSPROGRESS" for metaschema tables.
- DeleteValExp – an expression which is tested at the time of a record delete to see whether deletion will be allowed. Maps to a UML Tagged Value. From _File._Valexp.
- DeleteValMsg – the text of the error message when deletion is not allowed per DeleteValExt. Maps to a UML Tagged Value. From _File._Valmsg.
- DeleteValMsgStringAttr - string attributes for the DeleteValMsg. Maps to a UML Tagged Value. From _File._Valmsg-SA.
- AreaName – name of the database area for this table. Maps to a UML Tagged Value or an Extended Property in EA. From _Area._Area-Name via _StorageObject.

Note that, in databases with large numbers of tables, it is desirable when possible to group tables within packages according to primary use or subsystem (see «oeSubSystem» below). There is no structure in the OpenEdge database to define these clusters so they will have to be created manually or through some application specific code[7]. No special stereotype is defined for these packages.

«oeTableTrigger»

An «oeTableTrigger» is a body of code which fires in response to some record event. It maps to an UML Operation[8] on the corresponding «oeTable» or to a UML Component representing the trigger code (see below). Its properties are:

- oeTableTriggerGUID – unique identifier for the instance. Used in making links.
- oeTableGUID – unique identifier for the «oeTable» to which this trigger applies.
- TriggerType – one of the possible trigger types such as CREATE or WRITE. Maps to the name of the UML Operation and the Precondition of that Operation. From _File-Trig._Event.
- Override – a flag indicating whether it is possible to override this trigger. Maps to a UML Tagged Value. From _File-Trig._Override.
- oeProgramGUID – the unique identifier for the «oeProgram» identified as the code to run when the event occurs. See discussion. From _File-Trig._Proc-Name.

There is some question as to how this should be best mapped to UML. The relational structure in the metaschema provides a procedure name which is connected at run time to specific code with a CRC check to insure that it is the expected code. But, as a UML Operation, the code should actually be in the Operation itself. Since the initial implementation will not have the code in the model, the initial implementation, therefore, will create an «oeTableTriggerLink» between the «oeTable» and an «oeTableTrigger» which is a UML Component representing the code pointed to by the database.

«oeColumn»

An «oeColumn» is a field in a database table and maps to a UML Attribute on the corresponding «oeTable» with properties:

- oeColumnGUID – a unique identifier for the instance.
- oeTableGUID – the unique identifier for the associated «oeTable»..
- ColumnCanRead – a list of users authorized to read the column. Maps to a UML Tagged Value. From _Field._Can-Read.

[7] Examples of how these associations might be computationally derived include application suites where all tables in a particular package have a common prefix or where relevant package affiliation is noted in the description. Alternatively, this kind of grouping may have been previously made manually in an ER diagramming tool such as ERwin. These would be suitable for organizing the tables into Packages in UML as well.

[8] In Enterprise Architect there is an existing stereotype for «trigger» that is a stereotype of Operation, but since the code will not initially be included in the model, this structure will not be used. When and if we do import the code, the "oeTableTrigger» stereotype would be based on the «trigger» stereotype.

- ColumnCanWrite – a list of users authorized to write the column. Maps to a UML Tagged Value. From _Field._Can-Write.
- CharacterSet – the name of the character set of this column. Maps to a UML Tagged Value. From _Field._Charset.
- ColumnLabel – the label to use at the head of a column containing this value. Maps to a UML Tagged Value. From _Field._Col-label.
- ColumnLabelStringAttr – the string attributes applying to the ColumnLabel. Maps to a UML Tagged Value. From _Field._Col-label-SA.
- Collation – the name of the collation table to use with this column. Maps to a UML Tagged Value. From _Field._Collation.
- DataType – type of this column. Maps to the Data Type of the UML Attribute. From _Field._Data-type.
- Decimals – the number of decimal places in a Decimal data type. Maps to a UML Tagged Value. From _Field._Decimals.
- Description – a description of the attribute. Maps to the Notes of the UML Attribute. From _Field._Desc.
- Extent – the number of entries in an array datatype. Maps to a UML Tagged Value. From _Field._Extent.
- Name – the name of this column. Maps to the name of the UML Attribute. From _Field._Field-Name.
- Positon – the "r-code position" of the column. Maps to a UML Tagged Value. From _Field._field-rpos.
- CaseSensitive – a flag indicating whether the field is case sensitive. Maps to a UML Tagged Value. From _Fld-case.
- Format – the default display format for the column. Maps to a UML Tagged Value. From _Field._Format.
- DefaultFormatStringAttr – the string attributes of the default format. Maps to a UML Tagged Value. From _Field._Format-SA.
- Help – the default help text for the field. Maps to a UML Tagged Value. From _Field._Help.
- HelpStringAttr – the string attributes for the default help text for the field. Maps to a UML Tagged Value. From _Field._Help-SA.
- Initial – the initial value for the Attribute. Maps to the Initial Value for the Attribute. From _Field._Initial.
- InitialStringAttr – the string attributes for the initial value for the field. Maps to a UML Tagged Value. From _Field._Initial-SA.
- SideLabel – the label to use at the side of a field containing this value. Maps to a UML Tagged Value. From _Field._Label.
- SideLabelStringAttr – the string attributes applying to the SideLabel. Maps to a UML Tagged Value. From _Field._Label-SA.
- Mandatory – a flag indicating whether or not the field is nullable; mandatory = non-null. Maps to Not Null of UML Attribute. From _Field._Mandatory.
- Order – the logical sequence of the column within the table. Maps to a UML Tagged Value. From _Field._Order.
- NoDefaultDisplay – A flag indicating whether the field should be excluded from any default displays of the table; i.e., a "system field" is excluded from default display. Maps to a UML Tagged Value. From _Field._sys-field.
- ValExp – A expression which must evaluate to true for the field to be valid. Maps to a UML PostCondition. From _Field._Valexp.

- ValMsg – the message to display when ValExp is not true. Maps to a UML Tagged Value. From _Field._Valmsg.
- ValMsgStringAttr – the string attributes applying to the ValMsg. Maps to a UML Tagged Value. From _Field._Valmsg -SA.
- ViewAsPhrase – the default value to use in a View-As phrase for this field. Maps to a UML Tagged Value. From _Field._View-As.
- Width – the width of the field to use for SQL queries. Maps to a UML Tagged Value. From _Field._Width.
- IsPK – a flag indicating whether or not the column participates in the primary key for the table. Maps to the Primary Key flag on the UML Attribute. From _Index._Field-recid when _Field._Prime-Index = recid(_Index) and _Index-Field._Index-recid = recid(_Index) on the primary index.

Since we will be defining the actual indexes and those will be marked unique or not according to the index, we will not attempt to mark the uniqueness of the columns in the primary key[9].

«oeIndex»

An «oeIndex» is an ordered collection of columns which can be used for sorted and selective access to the table and which may provide a uniqueness constraint. It will be mapped to a UML Operation and its properties are:

- oeIndexGUID – a unique identifier for the instance.
- oeTableGUID – the unique identifier for the table to which this index applies. Used for making the association.
- IndexColumns – identifies the table columns used in the index; from _Index-_Field-recid; maps to Parameters of the Operation.
- ColumnSortOrder – a list of Ascending/Descending to match the IndexColumns; from Index_Field._Ascending; will be identified with a stereotype.
- ColumnAbbreviate – a list of Yes/No Abbreviate to match the IndexColumns; from Index_Field._Abbreviate will be identified with a stereotype.
- IsUnique – indicates whether the index is unique; from _Index._Unique; maps to a PostCondition on the Operation.
- IsWordIndex – a flag indicating that this field is a character field which is the basis for a primary index and is a word index. From _Index._Wordidx.
- IsPrimary – indicates whether this is the primary index for this table.

Note that since _Index-Field._Abbreviate is deprecated, but is in use in some databases, whether or not actually used in the code. Since the columns will be mapped to Parameters of the Operation and there are no appropriate flags for Sort Order or Abbreviate, we will identify these properties through the use of stereotypes of «Asc» = Ascending, not Abbreviate; «Desc» = Descending, not Abbreviate; «AscAbv» = Ascending, Abbreviate; «DescAbv» = Descending, Abbreviate.

Handling of Foreign Keys

Since a Progress database does not include explicit metadata on table joins such as one might wish for defining foreign keys, it is necessary to discover these in some other way. One would like to think that the indexes defined on a table would be a good indicator, but there is no guarantee that an index that is defined is actually used and certainly no guarantee that every file access is supported by an index. This specification does provide for capturing the empirical table joins from

[9] There are known to be some issues in this area in Enterprise Architect, but specifying the full index is a more complete specification anyway.

the code, but it seems desirable to also be able to indicate simple foreign key relationships directly. Depending on the naming standards of the database, it can be possible to "discover" these relationships by looking for identical column names, e.g., Order.CustID and Customer.CustID, but in many cases this is unreliable or problematic. Among the obstacles to simple discovery are simple naming differences, e.g., Customer.ID equivalent to Order.CustID; common column names in many tables creating false joins, e.g., X.CreateDate or X.UserID; use of table prefixes on column names so that every column has a unique name; and simple cases of necessary difference in naming, e.g., X.ParentID pointing to X.ID.

While there may or may not be an easy way to determine foreign key relationships other than by reference to empirical «*DataLink» relationships (see below) we will provide the «oeFK» and «oeFKLink» stereotypes for loading foreign key relationships when these are known by whatever means.

«oeFK» and «oeFKLink»

An «oeFK» is an indication in one table that it references another table and is mapped to a UML Operation on the Source Table. Its properties are:

- oeForeignKeyGUID – a unique identifier of the instance.
- oeSourceTableGUID – the unique identifier of the source table; used for making the association.
- oeSourceTableGUID – the unique identifier of the target table; used for making the association.
- oeForeignTableColumns – the list of Columns in the Target Table; mapped to Parameters of the UML Operation.

These values will be used to create an Operation in the Source table named FK_<Source table name>_<Target table name>. The «oeFKLink» will also be created as a directed UML Association from the Source to the Target.

«oeSequence»

An «oeSequence» is an instance of a database sequence. While one of the virtues of an OpenEdge sequence is that it is independent of tables and locking mechanisms, we are provisionally mapping it to a UML Class for each individual instance with a single Attribute having the same name as the Class. Its properties are:

- oeSequenceGUID – unique identifier for the instance.
- oeDatabaseGUID – the unique identifier for the database of which this sequence is a part. Used for link.
- SeqName – the name of the sequence. Maps to the name of the UML Class. From _Sequence._Seq-Name.
- SeqNumber – a number indicating the order of the sequence within the database. Maps to a UML Tagged Value. From _Sequence._Seq-Num.
- SeqInitial – a number indicating initial value of the sequence. Maps to a UML Initial Value of the Attribute. From _Sequence._Seq-Init.
- SeqIncrement– a number indicating increment value of the sequence. Maps to a UML Tagged Value. From _Sequence._Seq-Incr.
- SeqMin – a number indicating lower value of the sequence. Maps to a UML Lower Bound on the Attribute. From _Sequence._Seq-Min.
- SeqMax – a number indicating upper value of the sequence. Maps to a UML Upper Bound on the Attribute. From _Sequence._Seq-Max.

- SeqCycleOnLimit – a flag indicating whether the sequence should restart at the lower bound when it reaches the upper limit. Maps to a UML Tagged Value. From _Sequence._Cycle-Ok.
- Owner – a field indicating the id of the sequence owner. Maps to a UML Tagged Value. From _Sequence._Seq-Owner. Will normally be "PUB".

Profile for ABL Code

While the structure of a Progress database is quite formal and similar in most respects to other databases, ABL code is not only less formally structured, but individual applications are likely to be structured in very different ways. Therefore, the first problem in defining a profile for ABL code is in deciding what are the components and their potential relationships.

One obvious and unambiguous classification is the "compile unit", i.e., some body of code which when compiled becomes an ABL .r file[10]. Regardless of whether the extension of the source file which becomes this compile unit was .cls, .p, .w, or even missing, the compiled program will be a .r file and is the basis for the runtime system. Thus «oeCompileUnit» suggests itself strongly as the basis of the deployed system and a strong grouping focus for the source that composes it.

Clearly, there is a very significant difference in modern versions of ABL between the characteristics of those compile units which arise from a Class and all other compile units, so it seems important to make this distinction in the source files associated with compile units. While there is a convention in tools and examples from Progress Software of distinguishing between a .w file as a compile unit which contains a user interface and a .p file as one that does not, it is common in existing legacy applications to have .p files with user interfaces and for there to be many other variations such as different file extensions or missing extensions so that making any other distinction based on extension a part of the stereotype seems dangerous or even deceptive. Therefore, we will only distinguish between «oeProgram» and «oeClass» where the latter is any post 10.1A compile unit based on a Class definition and the former is any other compile unit.

ABL has the ability to execute blocks of code smaller than a complete compile unit once the compile unit itself has been instantiated as a persistent part of a session. Within an «oeClase» these are blocks are identified by a Method definition and those we shall associate with the stereotype «oeMethod». The expectation is that «oeClass» and «oeMethod» will be very close to the underlying UML Class and Operation, but the stereotypes are still worth defining in order to map closely to the specifics of the ABL implementation. Within an «oeProgram» the blocks are identified by either Procedure or Function definitions so we will associate these with «oeIP» (for Internal Procedure) and «oeFunction» stereotypes. Tentatively, a trigger block or `ON` block will be considered the same as a Procedure block, but with a Precondition[11]. We will use the term «oeSubCompileUnit» to refer to these three collectively. Note that while one might be inclined to model «oeSubCompileUnit»s as Operations within a Class, their role in ABL programming is often quite different from the integrated nature of Methods of a Class, so we will be modeling them as separate Components.

Since there are important aspects of the behavior of an «oeClass» which relate to the package with which it is associated, we will also define a stereotype for «oePackage». «oeInterface» will be used for a .cls file which is defined as an interface.

[10] Even if one deploys uncompiled source code, this code is compiled at run time before it is executed and becomes a virtual .r file.
[11] In the initial implementation, `ON` blocks will not be separated from the main body of the code as suggested here, but this may be added in a later version.

There are two ways in which one compile unit can instantiate, persistently or temporarily, a new instance of an «oeProgram». An «oeProgram» or, less commonly, an «oeClass» can `RUN` another «oeProgram» and we will identify this link with the stereotype «oeRunLink». An «oeClass» or, less commonly, an «oeProgram» can `NEW` another «oeClass» and we will identify this link with the stereotype «oeNewLink».

Execution of an «oeIP» by a given «oeCompileUnit» will be identified by a link with the stereotype «oeRunIPLink». Similarly, a reference to an «oeFunction» will be identified by the stereotype «oeRunFunction». A reference to an «oeMethod» will use the stereotype «oeMethodLink». Note that in all three cases the link is one between an «oeCompileUnit» or «oeSubCompileUnit» and the specified target.

In ABL, some operating system files are not themselves compile units, but are only included in other compile units. While typically identified by a .i file extension, we will rely on usage, not the name in identifying this type of unit. These will use the «oeInclude» stereotype and will be connected to the appropriate «oeCompileUnit» with the «oeIncludeLink» stereotype.

To connect «oeCompileUnit»s and «oeSubCompileUnits» with «oeTable»s we will use «oeCUDataLink*» and «oeSubCUDataLink*» to make the connections with trailing modifiers to indicate the nature of the access. These stereotypes will have properties to identify the columns accessed.

Details of these stereotypes will be provided in the following descriptions. In the properties for each stereotype there is a unique identifier with a name ending in GUID. This identifier will be used by the UML modeling tool to link together the various components. The specification for this GUID will be determined later with some experimentation with Enterprise Architect. One possible implementation might be `BASE64-ENCODE(GUID(GENERATE-UUID))`.

«oeCompileUnit» is documented below in structural and functional groupings.

<u>«oeProgram»</u>
An «oeProgram» is a stereotype for a UML Component. Every compile unit which is not a Class will be identified as an «oeProgram» with the following properties:

- oeProgramGUID – a unique identifier for this instance.
- Name – the name of the file as it would be used in a `RUN` statement. Maps to the name of the UML Class. See discussion below about «oePackage». If «oeProgram»s are included in «oePackage»s, then this name is simply the base name of the file.
- Description – where possible a brief description will be extracted from program headers. Maps to a Comment associated with the UML Component.
- PathName – the full path identifying the specific file from which the information has been extracted. Maps to a Tagged Value.
- HasUI – a flag indicating whether this compile unit has any user interface statements. Maps to a Tagged Value.
- HasDA – a flag indicating whether this compile unit has any database access statements. Maps to a Tagged Value.

- Code – the text of any code contained within this compile unit, excluding that contained in any subcompile units such as internal procedures or functions. Maps to the Code component of a Operation called MainBody[12].
- Parameters – any input or output parameters which are associated with executing this compile unit. Maps to Parameters on the Operation called MainBody.
- Attributes – all variables defined in this compile unit[13]. Note that a shared variable is essentially a public attribute. Maps to Attributes of the Component.

«oeIP»

An «oeIP» is a stereotype for a SubComponent of a UML Component. Each `PROCEDURE` block in an «oeProgram» will be identified as an «oeIP» with the following properties:

- oeIPGUID – a unique identifier for this instance. Used for linking with other components.
- oeProgramGUID – a unique identifier for the «oeProgram» containing this «oeIP». Provides a link to the UML Component.
- Name – the name of the procedure as it would be used in a `RUN` statement. Maps to the name of the UML SubComponent.
- HasUI – a flag indicating whether this subcompile unit has any user interface statements. Maps to a Tagged Value.
- HasDA – a flag indicating whether this subcompile unit has any database access statements. Maps to a Tagged Value.
- Code – the text of any code contained within this subcompile unit,. Maps to the Code component of an Operation with the same name as the «oeIP».
- Parameters – any input or output parameters which are associated with executing this procedure. Maps to Parameters on the Operation.
- Attributes – all variables defined in this subcompile unit. Maps to Attributes of the Component.

Note that some consideration needs to be given as to whether the use of `RETURN <value>` should constitute an effective Return Value for the Operation.

A trigger block will receive an additional value in the form of a Precondition.

«oeFunction»

An «oeFunction» is a stereotype for an Operation of a UML Component.. Each `FUNCTION` block in an «oeProgram» will be identified as an «oeFunction» with the following properties:

- oeFunctionGUID – a unique identifier for this instance. Used for linking with other components.
- oeProgramGUID – a unique identifier for the «oeProgram» containing this «oeFunction». Provides a link to the UML Component.
- Name – the name of the Function as it would be used when called. Maps to the name of the UML Component.
- HasUI – a flag indicating whether this subcompile unit has any user interface statements. Maps to a Tagged Value.
- HasDA – a flag indicating whether this subcompile unit has any database access statements. Maps to a Tagged Value.
- Code – the text of any code contained within this subcompile unit,. Maps to the Code component of an Operation with the same name as the «oeFunction».

[12] In the initial implementation, no code will not be imported into the model, but will be kept externally in Analyst or the filesystem and automation will be used to link from the model to the code, but this definition is retained for future use or for use by those who are using different tools.

[13] Initial implementation will include only shared variables.

- ReturnValueType – the type of the return value of the Function. Maps to the Return Value of the Operation.
- Parameters – any input or output parameters which are associated with executing this Function. Maps to Parameters on the Operation.

«oeInclude»
An «oeInclude» is a separate file which is not compilable and which is included in one or more compile units by reference. It is tentatively mapped to an UML Component. Its properties are:
- oeIncludeGUID – a unique identifier for this instance.
- Name – the name of the file as it would be used in an include reference. Maps to the name of the UML Component. See discussion below about «oePackage». If «oeInclude»s are included in «oePackage»s, then this name is simply the base name of the file.
- PathName – the full path identifying the specific file from which the information has been extracted. Maps to a Tagged Value.
- Description – where possible a brief description will be extracted from program headers. Maps to a Comment associated with the UML Component.
- HasUI – a flag indicating whether this file has any user interface statements. Maps to a Tagged Value.
- HasDA – a flag indicating whether this file has any database access statements. Maps to a Tagged Value.
- Code – the text of any code contained within this file. Maps to the Code component of the Operation MainBody.
- Parameters – any input or output argument which are associated with this file. Maps to Parameters on the Operation MainBody.

«oePackage»
An «oePackage» is a collection of «oeClass»s found in a common directory and is identified by the package portion of the Class name in combination with the USING specification. It maps to a UML Package. Its properties are:
- oePackageGUID – a unique identifier for this instance. Used for linking with other components.
- Name – the name of the directory in which the Class files reside. Maps to the name of the UML Package.

A separate «oePackage» layer will be constructed for each level in the hierarchy[14].

«oeInterface»
An «oeInterface» is a .cls file which is defined and used as an interface, i.e., it begins with the `INTERFACE` keyword. It is mapped to a UML Interface. Its properties are:
- oeInterfaceGUID – a unique identifier for this instance. Used for linking with other components.
- oePackageGUID – a unique identifier for the Package instance to which this interface belongs.
- Name – the basename of the .cls file. Maps to the name of the UML Interface.
- Description – where possible a brief description will be extracted from program headers. Maps to a Comment associated with the UML Component.

[14] Note that one could and possibly should create "oePackage» layers for "oeProgram»s as well as «oeClass»s, but AutoEdge represents a case where this would present some difficulties because of the practice in AutoEdge of having `PROPATH` components which nest, thus putting some given files along more than one path sequence from the `PROPATH`. This might be resolvable by noting which references are actually used in the code. E.g., if there is a file a/b/c/x.p and the `PROPATH` contains both a/b and a/b/c, then x.p will either be reference in the code as run x.p or run c/x.p and that could determine which sequence was used for the packages. Of course, it is possible that both would be found, but one hopes that is not the case. Ordinary, non-stereotyped Packages will be used to organize non-Class components, typically based on the directory structure in which files are located beneath the `PROPATH` roots.

- PathName – the full path identifying the specific file from which the information has been extracted. Maps to a Tagged Value.
- Attributes – all variables and properties defined in this compile unit. Maps to Attributes of the Class.

«oeClass»
An «oeClass» is a .cls file which is defined and used as a Class, i.e., it begins with the CLASS keyword. It is mapped to a UML Class. Its properties are:
- oeClassGUID – a unique identifier for this instance. Used for linking with other components.
- oePackageGUID – a unique identifier for the Package instance to which this Class belongs.
- Name – the basename of the .cls file. Maps to the name of the UML Class.
- Description – where possible a brief description will be extracted from program headers. Maps to a Comment associated with the UML Component.
- PathName – the full path identifying the specific file from which the information has been extracted. Maps to a Tagged Value.
- Attributes – all variables and properties defined in this compile unit. Maps to Attributes of the Class.

Since no code is allowed in the main body of an ABL Class, it is assumed that HasUI and HasDA cannot apply to the class independent of its methods.

«oeMethod»
An «oeMethod» is a block of code within an «oeClass» file, delimited by a METHOD…END block, including any CONSTRUCTOR…END and DESTRUCTOR…END blocks. It is mapped to a UML Operation. Its properties are:
- oeMethodGUID – a unique identifier for this instance. Used for linking with other components.
- oeClassGUID – the unique identifier for the oeClass or oeInterface to which this method belongs.
- Name – the name by which the method is invoked. Maps to the name of the UML Operation.
- HasUI – a flag indicating whether this method has any user interface statements. Maps to a Tagged Value.
- HasDA – a flag indicating whether this method has any database access statements. Maps to a Tagged Value.
- Code – the text of any code contained within this method. Maps to the Code component of the Operation.
- ReturnValue – the type of the return value of the Method.
- Parameters – any input or output parameters which are associated with executing this Method. Maps to Parameters on the Operation.
- Attributes – all variables and properties defined in this compile unit. Maps to Attributes of the Class.

Note that UML Attributes do not contain code, but ABL Properties may contain code, even though they are primarily attributes. Therefore, a convention will be required for creating Getter and Setter method names associated with any property that contains code in order to have an appropriate place to put the code.

«oeRunLink»
An «oeRunLink» is a connection between an «oeCompileUnit» or «oeSubCompileUnit» and a «oeProgram» based on the RUN statement, i.e., the operation which instantiates the «oeProgram». It is mapped to a UML Association. Its properties are:

- oeRunLinkGUID – a unique identifier for this instance.
- oeSourceGUID – the unique identifier for the «oeCompileUnit» or «oeSubCompileUnit» in which the RUN statement appears. Maps to the source UML Component in the Association.
- oeTargetGUID – the unique identifier for the «oeProgram» which is RUN. Maps to the target UML Component in the Association.
- IsPersistent – a flag indicating whether the connection is RUN PERSISTENT. Maps to a UML Tagged Value.
- IsSuper – a flag indicating whether the connection is to a superprocedure. Maps to a UML Tagged Value.
- oeParameterN – a documentation of the value given to each parameter referenced in the RUN statement. Maps to UML Tagged Values.

It is not immediately clear what should be done with a session super since it is invoked by one «oeCompileUnit», but might be used by many. Each usage will have an «oeRunIPLink», of course, but it is possible that each «oeCompileUnit» containing any «oeRunIPLinks» to session supers should also be provided with an Association link.

«oeRunIPLink»

An «oeRunIPLink» is a connection between an «oeCompileUnit» or an «oeSubCompileUnit» and an «oeIP» based on either RUN…IN or a RUN of an IP in a SUPER. It is mapped to a UML Association. Its properties are:

- oeRunIPLinkGUID – a unique identifier for this instance.
- oeSourceGUID – the unique identifier for the «oeCompileUnit» in which the RUN statement appears. Maps to the source UML Class in the Association.
- oeTargetIPGUID – the unique identifier for the «oeIP» which is RUN. Maps to the target UML Class in the Association.
- oeParameterN – a documentation of the value given to each parameter referenced in the RUN statement. Maps to UML Tagged Values.

«oeRunFunctionLink»

An «oeRunFunctionLink» is a connection between an «oeCompileUnit» or «oeSubCompileUnit» and an «oeFunction» based on a forward reference to a Function in a different «oeCompileUnit». It is mapped to a UML Association. Its properties are:

- oeRunFunctionLinkGUID – a unique identifier for this instance.
- oeSourceGUID – the unique identifier for the «oeCompileUnit» in which the RUN statement appears. Maps to the source UML Class in the Association.
- oeTargetFunctionGUID – the unique identifier for the «oeFunction». Maps to the target UML Class in the Association.
- oeParameterN – a documentation of the value given to each parameter referenced in the FUNCTION call. Maps to UML Tagged Values.

«oeIncludeLink»

An «oeIncludeLink» is a connection between an «oeCompileUnit» or «oeSubCompileUnit» and an «oeInclude» based on a include reference. It is mapped to a UML Assembly connection. Its properties are:

- oeIncludeLinkGUID – a unique identifier for this instance.
- oeSourceGUID – the unique identifier for the «oeCompileUnit» in which the RUN statement appears. Maps to the source UML Class in the Assembly.

- oeTargetIncludeGUID – the unique identifier for the «oeInclude». Maps to the target UML Class in the Assembly.
- oeArgumentN – a documentation of the value given to each argument referenced in the include reference. Maps to UML Tagged Values.

«oeNewLink»
An «oeNewLink» is a connection between an «oeCompileUnit» or «oeSubCompileUnit» and a «oeClass» based on the NEW statement, i.e., the operation which instantiates the «oeClass». It is mapped to a UML Association. Its properties are:
- oeNewLinkGUID – a unique identifier for this instance.
- oeSourceGUID – the unique identifier for the «oeCompileUnit» or «oeSubCompileUnit» in which the NEW statement appears. Maps to the source UML Class in the Association.
- oeTargetClassGUID – the unique identifier for the «oeClass» which is NEW. Maps to the target UML Class in the Association.
- oeParameterN – a documentation of the value given to each parameter referenced in the NEW statement. Maps to UML Tagged Values.

«oeMethodLink»
An «oeMethodLink» is a connection between an «oeCompileUnit» or «oeSubCompileUnit» and an «oeMethod» based on the invocation of a method in an «oeClass». It is mapped to a UML Association. Its properties are:
- oeMethodLinkGUID – a unique identifier for this instance.
- oeSourceUnitGUID – the unique identifier for the «oeCompileUnit» or «oeSubCompileUnit» in which the «oeMethod» invocation statement appears. Maps to the source UML Class in the Association.
- oeTargetMethodGUID – the unique identifier for the «oeMethod» which is invoked. Maps to the target UML Class containing the «oeMethod» in the Association.
- oeParameterN – a documentation of the value given to each parameter referenced in the Method invocation. Maps to UML Tagged Values.

«oeInheritanceLink»
An «oeInheritanceLink» is a connection between two «oeClass»es in which one inherits the other. It is mapped to a UML Generalization Link. Its properties are:
- oeInheritanceLinkGUID – a unique identifier for this instance.
- oeChildClassGUID – the unique identifier for the «oeClass» which is the child. Maps to the UML Class which is the child in the Generalization link.
- oeParentClassGUID – the unique identifier for the «oeClass» which is the parent. Maps to the UML Class which is the parent in the Generalization link.

«oeImplementationLink»
An «oeImplementationLink» is a connection between an «oeInterface» and an «oeClass» which implements it. It is mapped to a UML Dependency Link. Its properties are:
- oeImplementationLinkGUID – a unique identifier for this instance.
- oeClassGUID – the unique identifier for the «oeClass» which is the implementor. Maps to the UML Class in the Dependency link.
- oeInterfaceGUID – the unique identifier for the «oeInterface» which is the interface being implemented. Maps to the UML Class which is the interface in the Dependency link.

Profile for Links Between ABL Code and Database Elements

In addition to connections among database elements and connections among code elements, it is desirable to connect code with database elements where they are referenced. These links will be made between a «oeCompileUnit» or «oeSubCompileUnit» and an «oeTable» with a field list to indicated which «oeField»s are referenced in that particular body of code and a list of Access Modes to indicate what kind of access is made. Access Modes will be Like, ReadNoLock, ReadSharedLock, ReadExclusiveLock, Update, Create, and Delete.

«oeDataLinkReadOnly» and «oeDataLinkModify»

«oeDataLinkReadOnly» and «oeDataLinkModify» are a connections between an «oeCompileUnit» or «oeSubCompileUnit» and an «oeTable». «oeDataLinkReadOnly» is used for `LIKE` and `NO-LOCK` reads in which no data is modified and «oeDataLinkModify» is used for create, update, and delete operations. They are mapped to a UML Dependency link. Its properties are:

- oeDataLinkGUID – a unique identifier for the instance.
- oeSourceUnitGUID – the unique identifier for the «oeCompileUnit» or «oeSubCompileUnit» containing the reference. This maps to the UML Component which is the Source in the Dependency relationship.
- oeTableGUID – the unique identifier for the «oeTable» which is referenced. This maps to the UML table corresponding to the Target Class in the Dependency relationship.
- ColumnList – a list of «oeColumn»s included in the reference. Maps to a UML Tagged Value.
- AccessModeList – a list of Access Modes which apply to this reference as above. Maps to a UML Tagged Value.
- WhereClause – all `WHERE` clauses in the unit which relate to this link. Maps to a UML Tagged Value, one per instance.

Profile for Structural and Functional Groupings

In any given application, there are logical groupings which are useful in analysis which cluster together related functions. The primary structuring principle common to most applications is that represented by the menu system which provides access to individual functions. Some applications also have batch functions not accessed from a menu, but these can be grouped similarly by the use of virtual menus. The menu structure is an important structure of the user interface and provides insight in the functional structure as well.

For the menu structure, the concept is to create «oeMenu» packages for each menu. This package may contain other «oeMenu» packages or «oeMenuItem», which are non-menu selections mapped to Components. When an «oeMenu» or «oeMenuItem» appears on more than one menu, it will be moved to a Shared package at the level of the common parent. Since package contents are therefore not necessarily complete, the complete menu will appear in the Notes of each «oeMenu». Each «oeMenuItem» will be linked to an «oeFunctionalUnit» which is mapped to a Package and contains one or more «oeCompileUnit»s. «oeCompileUnit»s are also packages which contain the «oeProgram» or «oeClass» and any «oeInclude»s used by the program or class.

«oeMenu»

An «oeMenu» is a component which corresponds to a menu presented to a user. Where there are hierarchical menus, multiple «oeMenu»s will be nested within each other to correspond to the hierarchy of menus available to the user. It will be mapped to a UML Package and its properties are:

- oeMenuGUID – a unique identifier for this instance.

- oeMenuName – an appropriate name to identify the menu.
- oeMenuList – a text corresponding to the menu presented to the use. It is mapped to the Notes of the Component.

When there are batch processes that are not launched from menus, a virtual menu can be created in order to represent all functions in a system. When an «oeMenu» has multiple parents, it will be promoted to the level of the common parent and placed in a Shared package.

«oeMenuItem»

An «oeMenuItem» is an entry on a menu which is not itself a menu, e.g., one that runs a program. It is mapped to a UML Component and its properties are:

- oeMenuItemGUID – a unique identifier for this instance.
- oeMenuItemName – an appropriate name to identify the menu item, e.g. the text of the line in the menu.

«oeMenuLink»

An «oeMenuLink» is a connection between an «oeMenu» and one or more other «oeMenu»s or «oeMenuItem»s. It will be mapped to a UML Nesting relationship and its properties are:

- oeMenuLinkGUID – a unique identifier for the instance.
- oeSourceMenuGUID – the unique identifier for the «oeMenu» which is the source of the link. Maps to the UML Source in the link.
- oeTargetGUID – the unique identifier for the «oeMenu» or «oeMenuItem» which is the target of the link. Maps to the UML Target in the link.

«oeFunctionalUnit»

An «oeFunctionalUnit» is a collection of code components which work together to perform some function such as that corresponding to selection of a menu item. There is a single start point and control returns to the caller when complete. A similar standard will be used to apply to batch processes. It is mapped to a UML Package and its properties are:

- oeFunctionalUnitGUID – a unique identifier for the instance.
- Name – A name for the functional unit such as the text of the Menu Item without any menu numbers.

The Notes of the «oeFunctionalUnit» will contain a list of all «oeCompileUnits which it contains. One of these will be designated as Primary, indicating which one is actually run by the menu. Those within the same package will be shown only with their basename while those from other packages will indicate the source package. «oeFunctionalUnits will be grouped into Packages according to the structure of the Primary compile unit (see below).

«oeMenuFunctionLink»

An «oeMenuFunctionLink» is a connection between an «oeMenuItem» and the corresponding «oeFunctionalUnit. It is mapped to a Dependency relationship and its properties are:

- oeMenuFunctionLinkGUID – an unique identifier for the instance.
- oeMenuItemGUID – the unique identifier for the «oeMenuItem».
- oeFunctionalUnitGUID – the unique identifier for the «oeFunctionalUnit»

«oeCompileUnit»

An «oeCompileUnit» is any .r, regardless of the type of source file from which it derives. It maps to an UML Package since it is a container for the source components. It's properties are:

- oeCompileUnitGUID – a unique identifier for the instance.

- Name - The name of the file as it would be used in a RUN statement. Maps to the name of the UML Package. See discussion above about «oePackage». If «oeProgram»s are included in «oePackage»s, then this name is simply the base name of the file.

«oeCompileUnits» will be organized into Packages according to the directory structure used to store them in a deployed system.

«oeFunctionCULink»
An «oeFunctionCULink» is a connection between an «oeFunctionalUnit» and the «oeCompileUnit»s which it contains. It is mapped to a UML Dependency relationship and its properties are:

- oeFunctionCULinkGUID – an unique identifier for the instance.
- oeFunctionalUnitGUID – the unique identifier for the «oeMenuItem».
- oeCompileUnitGUID – the unique identifier for the «oeFunctionalUnit»

«oeCUContentsLink»
An «oeCUContentsLink» is a connection between an «oeCompileUnit» and the «oeProgram», «oeClass» and any «oeInclude»s which compose it. It is mapped to a UML Dependency relationship and its properties are:

- oeCUContentsLinkGUID – an unique identifier for the instance.
- oeCompileUnitGUID – the unique identifier for the «oeFunctionalUnit»
- oeContentsGUID – the unique identifier for the «oeCompileUnit» and the «oeProgram», «oeClass» or «oeInclude»s.

Profile for User Interface
In a later version of this Profile we will explore the options for capturing the layout of the user interface using PRO/Dox or a similar tool.

Action Items
1. Complete section on User Interface.
2. Create separate document for XMI Specification corresponding to Profile.
3. Create XML file to load stereotype definitions.
4. Create EA Profile
5. Create sample EA project

Areas Not Yet Scheduled for Handling
1. Full information corresponding to structure file.
2. Codepage and collation information.
3. All Dataserver information.

Revision History

Revision	Date	Nature of Revision
0.1	27 May 2007	Initial draft with minimal database & no user interface.
0.2	1 June 2007	Addition of initial set of database components
0.3	26 June 2007	Add Functional and Structural elements
0.4	12 November 2007	General overall revision for tentative implementation