# Grammar and API for Rational Rose Petal files

M. Dahm

July 19, 2001

Version 1.1

## 1 Introduction

Rational Rose [2] is probably the most widely used tool for software engineering processes, although it has several deficiencies (which we will not discuss here). In the CRAZYBEANS project I tried to tackle one of them related to the "roundtrip engineering" process. One can import source files into Rose and create class diagrams from them. Vice versa one can create classes (and IDL specifications or whatever you need) from class (and other) diagrams. Yet these creation processes can not be influenced by the user in a convenient, flexible and extensible way. Another problem is that these models are proprietary and some people would like to have the possibility to write a converter for these models into a different format used by another tool.

Every application developed and also every company follows different policies and may need different conventions for the generated code. So it is clearly desirable for them to influence the way in which diagrams and other data are mapped to source code and vice versa. For example, one company might have the convention to attach notes to classes and associations containing OCL [1] statements (or whatever). Rose wouldn't not not regard them when creating code, but a tool specifically adapted for this need could do so.

So how can all this be accomplished? Well, the easiest way is to have an API that is capable of reading and writing the files created by Rational Rose, called "Petal files" (ending with `.mdl` or `.ptl`) which contain the given model of the application. The file format is undocumented, I'll try to explain it in the following sections anyway. The CRAZYBEANS API to read and write these files is explained in section 5.

### 1.1 Petal file format

The format of petal files generated by Rational Rose (ending with `.mdl` or `.ptl`) is not documented, yet it is an ASCII format, fortunately. We have been able to match several entities of its contents to what you see on the screen. Yet we do not understand all of it and some parts of the file seem not to be of general interest. I focused mostly on interpreting class diagrams, though use case und other diagrams can be analyzed, too.

The file format looks roughly like a lisp data structure, consisting of several nested levels enclosed in parentheses which form a *tree* of nodes. Take a look at one of your files yourself to see what I mean. The main data structure are "objects", i.e., items like

```
(object Petal
   version     42
   _written    "Rose 4.5.8163.3"
   charSet     0)
```

These objects always have a name ("Petal" in the example), optional parameter strings and an optional tag. The general form of is shown in section 2.2. There is quite a number of different

kinds objects. We mapped them to Java classes which can be found in the package `cb.petal`. The file format itself is not very "object-oriented", but we tried to use subclassing wherever it made sense. All objects have a number of properties which are mapped to set/get methods in the Java classes. A special property is the "quid" (which probably stands for "qualified identifier"), a globally unique identfier (coded as a hexadecimal number) for an object. References to other objects (e.g. an associaton) are defined by the "quidu" property.

There are two parts of a petal file which are of general interest for us: One section specifies the data model (classes, associations, ...) and another one specifies the "views" for the model (there may be many). The views are what you see on the screen although the underlying model may contain more information. This model-view pattern might ring a bell for some readers... Views may also contain additional "notes" which are not part of the data model but can be thought of as some kind of comments.

The following section gives an (incomplete) BNF grammar for petal files.

# 2 The grammar

Identifiers in angle brackets denote non-terminals, text in double-quotes denotes strings, bold text denotes terminal symbols. Comments are written *italic* and start with two slashes. Not all existing petal objects are listed here, because there are too many. However, the generic form is shown in section 2.2 which all petal objects conform to. Please refer to the concrete Java files in package `cb.petal` for details of the adjustable properties. The order of properties does not matter in most cases (sometimes Rose seems to be picky). The ugly thing is that property names may occur multiply, e.g., an object may have multiple "label"s attached to it.

## 2.1 Petal nodes

The tree defined by the petal file may have the following kinds of nodes:

⟨PetalNode⟩ → ⟨Object⟩ | ⟨Literal⟩ | ⟨List⟩

⟨List⟩ → **(list** ⟨name⟩ ⟨Object⟩* **)**

⟨Literal⟩ → ⟨Value⟩ | ⟨Tuple⟩ | ⟨Tag⟩ | ⟨Location⟩ | ⟨stringliteral⟩ | ⟨int⟩ | ⟨boolean⟩ | ⟨float⟩

⟨Value⟩ → **(value** ⟨name⟩ ⟨stringliteral⟩ **)**

⟨Tuple⟩ → **(** ⟨string⟩ ⟨int⟩ **)**

⟨Tag⟩ → @⟨int⟩

⟨Location⟩ → **(** ⟨int⟩ **,** ⟨int⟩ **)**

All of them are leaf nodes except for Object and List. The literals will be defined in section 2.3.

## 2.2 General format of an object

An object always has a name and optionally a list of strings which I call parameters, and some markup the form `@2` that we call "tags". The first parameter is usually the name for that object, the class name, e.g.. The second parameter may be the fully qualified name of an object. Tags

are used in the context of views only and their purpose is to provide some local numbering mechanism.

Most objects have a "quid" property and can be either be identified by this unique number or by their fully-qualified name which is given by their position in the tree (e.g., "`Logical View::University::Professor`").

Objects just consist of a list of key-value pair of properties, property names may occur multiply in some case, i.e. are not necessarily unique within the list. We will usually not list all possible properties, but only those of special interest.

⟨Object⟩ → **(object** ⟨name⟩ ⟨string⟩* [⟨Tag⟩]
(⟨name⟩ ⟨PetalNode⟩)* **)**

### 2.2.1 Example

The following examples shows how a plain "Student"class derived from class "Person" is represented. It has three properties (it may be more in fact) where two of them refer to literals and one ("superclasses") refers to a list of other classes which are identified via the quidu property. This is a list, because languages like C++ allow multiple inheritance.

```
(object Class "Student"
   quid          "3AE987720329"
   superclasses  (list inheritance_relationship_list
                    (object Inheritance_Relationship
                       quid        "3AE9877B01D8"
                       supplier    "Logical View::University::Person"
                       quidu       "3AE987400197"))
   language      "Java")
```

## 2.3 Literals

Strings come in two flavors: The standard string with text between two double quotes and multiline strings where each line starts with |.

⟨multistring⟩ → ⟨newline⟩ (| ⟨text⟩)+
⟨string⟩ → **"**⟨text⟩**"**
⟨stringliteral⟩ → ⟨string⟩ | ⟨multistring⟩

Qualified names have the form `ClassCategory::Package::Class`, for example, "`Logical View::University::Professor`".

⟨qname⟩ → ⟨string⟩

The rest is pretty much standard.

⟨name⟩ → ⟨char⟩ (⟨char⟩ | ⟨digit⟩)*
⟨text⟩ → ⟨anychar⟩*
⟨ident⟩ → "⟨hexnumber⟩"
⟨number⟩ → ⟨digit⟩+
⟨hexnumber⟩ → (⟨digit⟩ | ⟨hexdigit⟩)+
⟨digit⟩ → **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**
⟨hexdigit⟩ → **A** | **B** | **C** | **D** | **E** | **F**
⟨int⟩ → **[-]** ⟨number⟩

⟨float⟩ → **[-]** ⟨number⟩ **.** ⟨number⟩
⟨boolean⟩ → **TRUE** | **FALSE**
⟨char⟩ → **a** | **b** | ... | **z** | **A** | **B** | ... | **Z**
⟨ws⟩ → ⟨blank⟩ | ⟨tab⟩

# 3 Petal file format

Starting from the root of the file the overall structure looks like this:

⟨petalfile⟩ → ⟨petal⟩ ⟨design⟩

⟨petal⟩ → **(object Petal**
**version** ⟨number⟩
**_written** ⟨ident⟩
**charSet** ⟨number⟩ **)**

⟨design⟩ → **(object Design** "Logical View"
**is_unit TRUE**
**is_loaded TRUE**
**quid** ⟨ident⟩
**defaults** ⟨Defaults⟩
**root_usecase_package** ⟨UseCaseCategory⟩
**root_category** ⟨LogicalCategory⟩
**root_subsystem** ⟨SubSystem⟩
**process_structure** ⟨Processes⟩
**properties** ⟨Properties⟩
**)**

## 3.1 Class categories

Now the interesting part is the logical view of the petal file (property root_category), because it contains all the class diagrams. Therefore we just explored this part in depth. The root_usecase_package contains use case diagrams, root_subsystem component diagrams, and process_structure contains sequence diagrams. Their structure is quite similar to the LogicalCategory.

⟨LogicalCategory⟩ → **(object Class_Category** "Logical View"
**quid** ⟨ident⟩
**exportControl** ⟨string⟩
...
**logical_models** ⟨List⟩ *// of Class Association and further ClassCategory objects*
**logical_presentations** ⟨List⟩ *// of ClassDiagram objects*
**properties** ⟨Properties⟩ **)**

⟨Properties⟩ → **(object Properties**
**attributes** ⟨List⟩ *// of attribute objects*
**)**

The first list specifies the possible contents of the class (and other) diagrams, classes and associations, in particular, and serves as a data model. It also may contain further LogicalCategory objects and the data model for interaction/sequence diagrams in so called Mechanism objects.

The second list specifies the views (in this case class diagrams) in which these classes and associations may appear(see section 4).

## 3.2 Class

A class object (i.e. a member of the first list) contains all the information you might expect in a class, encapsulated in further objects. It contains fields, methods, references to the superclass(es), and the implemented interfaces. Classes may also have an access modifier and a stereotype associated with them. Just the naming convention is a little bit different in Rose (in terms of usual Java naming conventions): Methods are operations, fields are class attributes, access modfiers are named "exportControl", and "implement" is called "realize".

⟨Class⟩ → **(object Class** ⟨string⟩ *// class name*
**quid** ⟨ident⟩
**operations** ⟨List⟩ *// of Operation objects*
**class_attributes** ⟨List⟩ *// of ClassAttribute objects*
**superclasses** ⟨List⟩ *// of InheritanceRelationship objects*
**used_nodes** ⟨List⟩ *// of UsesRelationship objects*
**realized_interfaces** ⟨List⟩ *// of RealizeRelationship objects*
**exportControl** ⟨string⟩
**language** ⟨string⟩
**stereotype** ⟨string⟩ **)**

For (Java) interfaces the convention is to have an "Interface" stereotype for the class.

### 3.2.1 Operation objects

⟨Operation⟩ → **(object Operation** ⟨string⟩ *// method name*
**quid** ⟨ident⟩
**parameters** ⟨List⟩ *// of formal Parameter objects*
**result** ⟨string⟩ *// return type*
**opExportControl** ⟨string⟩
**stereotype** ⟨string⟩ **)**

### 3.2.2 Class attributes

⟨ClassAttribute⟩ → **(object ClassAttribute** ⟨string⟩ *// attribute name*
**quid** ⟨ident⟩
**type** ⟨string⟩ *// attribute type*
**exportControl** ⟨string⟩
**stereotype** ⟨string⟩ **)**

### 3.2.3 Inheritance relationship

⟨InheritanceRelationship⟩ → **(object Inheritance_Relationship** ⟨string⟩
**supplier** ⟨string⟩ *// eg Logical View*: *:Person*
**quidu** ⟨ident⟩ *// ID of referenced class eg Person*
**quid** ⟨ident⟩ **)**

#### 3.2.4 Uses relationship

⟨UsesRelationship⟩ → **(object Uses_Relationship** ⟨string⟩
**supplier** ⟨qname⟩ *// eg Logical View*: :*Person*
**quidu** ⟨ident⟩ *// ID of used class eg Person*
**quid** ⟨ident⟩ **)**

#### 3.2.5 Realize relationship

⟨RealizeRelationship⟩ → **(object Realize_Relationship** ⟨string⟩
**supplier** ⟨qname⟩ **quidu** ⟨ident⟩ *// ID of implemented interface*
**quid** ⟨ident⟩ **)**

### 3.3 Associations

An association contains a list of (exactly two) roles, i.e. both ends of the association are described with their role name, cardinality, etc..

⟨Association⟩ → **(object Association** ⟨string⟩ *// association name*
**roles** ⟨List⟩ *// of Role objects*
**AssociationClass** ⟨qname⟩ *// of association class*
**quid** ⟨ident⟩ **)**

#### 3.3.1 Role objects

⟨Role⟩ → **(object Role** ⟨string⟩ *// role name*

**supplier** ⟨qname⟩ *// associated with the given class*
**quidu** ⟨ident⟩ *// ID of associated class*
**is_navigable** ⟨boolean⟩ *// navigable in both directions?*
**is_aggregate** ⟨boolean⟩ *// aggregation?*
**Containment** ⟨string⟩ *// shared aggregation?*
**client_cardinality** ⟨Value⟩ *// cardinality like 1...n*
**quid** ⟨ident⟩ **)**

If "is_navigable" is false, the association can only be read in one direction. "is_aggregate" specifies whether this is an aggregation (drawn as a rhomb), the "Containment" of the *other* role specifies whether it is a shared aggregation, i.e., possible values are "By value" and "By reference". The "client_cardinality" (obviously) specifies the cardinality of the role.

### 3.4 Mechanism objects

Mechanism object contain the abstract description of a sequence or collaboration diagram (which are semantically equivalent).
TODO

## 4 Class and other diagrams

A Rose model may contain one or more class diagrams (and other kinds of diagrams not described here). They are listed under the "logical_presentations" property of the class category "Logical View" (see section 3.1).

⟨ClassDiagram⟩ → **(object ClassDiagram** ⟨string⟩ *// class name*
**quid** ⟨ident⟩
**title** ⟨string⟩
**zoom** ⟨int⟩
**max_height** ⟨int⟩
**max_width** ⟨int⟩
**origin_x** ⟨int⟩
**origin_y** ⟨int⟩
**items** ⟨List⟩ *// of View objects*
**)**

## 4.1 View objects

There are several View objects that may be listed in the "items" list, in particular class views, association views (for some peculiar reason named AssociationViewNew), note views, etc.. In general there are view objects for eveything listed in the "logical_models" property of the "Logical View" (see section 3.1). View objects do have no quid property, but most of them have a tag.

### 4.1.1 ClassView objects

⟨ClassView⟩ → **(object ClassView** ⟨string⟩ ⟨string⟩ ⟨Tag⟩*// FQN and index*
**location** ⟨Location⟩
**quidu** ⟨ident⟩
**label** ⟨ItemLabel⟩
**stereotype** ⟨ItemLabel⟩
**)**

⟨ItemLabel⟩ → **(object ItemLabel**
**location** ⟨Location⟩
**Parent_View** ⟨Tag⟩ *// Tag of the class view*
**label** ⟨string⟩
**)**

The tag of the class view is a running number and the first string is always equal to "Class" and the second refers to the fully qualified name of the class. In the ItemLabel object the interesting properties are "label" which defines the text to be displayed and "Parent_View" which is the same tag as the surrounding class view.

### 4.1.2 Example

```
(object ClassView "Class" "Logical View::Mentor" @2
   location (304, 1152)
   label    (object ItemLabel
               Parent_View @2
               location    (172, 1145)
               label       "Mentor")
   stereotype (object ItemLabel
                 Parent_View @2
                 location    (172, 1095)
                 label       "<<Interface>>")
   icon           "Interface"
```

```
    icon_style      "Label"
    quidu           "3AE987AB0209"
    width           282)
```

#### 4.1.3 AssociationViewNew objects

Associations (3.3) are displayed with AssociationViewNew objects, there is nothing really new for them to the reader except that their labels are of a different type and that the roles of an association are mapped to according RoleView objects. It's probably best to see an example again:

```
(object AssociationViewNew "teach" @12
  location   (710, 720)
  label      (object SegLabel @13
    Parent_View @12
    location   (710, 661)
    font       (object Font
                 italics    TRUE))
  line_color 3342489
  quidu      "3AE988420057"
  roleview_list (list RoleViews
    (object RoleView "$UNNAMED$0" @14
      Parent_View @12
      location   (310, 0)
      quidu      "3AE988420332"
      client     @12
      supplier   @4
      line_style 0)
    (object RoleView "$UNNAMED$1" @15
      Parent_View @12
      location   (310, 0)
      quidu      "3AE988420333"
      client     @12
      supplier   @8
      line_style 0)))
```

#### 4.1.4 Other view objects

There several other view objects describing relations between objects which are very similar in pattern, in particular InheritView, RealizeView, AttachView, and UsesView. The general syntax of them is:

⟨OtherView⟩ → **(object** ⟨name⟩ ⟨Tag⟩
**quidu** ⟨ident⟩
**client** ⟨Tag⟩
**supplier** ⟨Tag⟩
)

For example, for an InheritView, the "quidu" would refer to the according InheritanceRelationship object (see 3.2.3). The "client" and "supplier" property refer to the tags of the class view objects that represent the class and its super class.

There are probably even more view objects we forgot to mention, NoteView, for instance, but they're quite similar.

# 5 The CrazyBeans framework

The file format is in fact not "object-oriented", though it is obviously targeted to describe object-oriented structures. This means that the hierarchy of classes and interfaces we developed for Crazy Beans is rather arbitrary. We probably made some wrong assumptions and we tested it only with the models we could get hold on. We tried to give the API some structure using subclassing and through interfaces.

There are probably cases where a petal object parsed from a file does not define a certain property although the API may claim so. We're out of luck here, because we only can verify that empirically. Sometimes worse the meaning of properties is often "overloaded", i.e., the property "stereotype" sometimes refers to a string (as one would assume), somtimes to a label, and may be even a boolean value.

This being said, it works quite well for me by now and ww hope it does for your purposes, too. distribution contains four packages which are briefly described in the following sections. Please refer to the distributed API documentation and the examples for details.

## 5.1 The Petal package

This part of the API contained in the package `cb.petal` contains all the nodes (we could find) that make up a Rose petal file. I.e, every entity found in a model is mapped to class. This is true as well for basic structures such as lists and literals as well as for "petal objects" (see 2.2). There are methods to set and remove properties.

Every petal object has an `accept()` method to be used in conjunction with the visitor pattern. There also some predefined visitors that use certain traversal strategies. You should subclass these instead of writing own visitor, because the visitor may change if new entities of interest are discovered.

Petal objects also contain a pointer to their containing parent in the tree formed by a petal file. They also have an `init()` method which is called after they have been added to model. This is used, e.g., to maintain lookup information. The root of the tree is called `PetalFile` and contains additional lookup methods, for example, to find an entity by its `quid` property or a class by its fully qualified name.

## 5.2 The parser package

The classes in `cb.parser` simply make up a parser for petal files, the parser is written using JavaCC [3]. It makes sure that all references (e.g. pointers to the parent node) are set up correctly and that `init()` gets called.

## 5.3 The utility package

This package obviously contains useful classes for the API. Of special interest might be the `PetalObjectFactory`, which allows to create petal objects with more comfort. It reads serialized template objects from `templates` directory. The factory does not call `init()` on the created objects, this is done when the objects have been added to the model.

## 5.4 The generator package

The framework defined here allows you to create classes (or whatever) from class diagrams. The two main classes are the `Generator` which mainly takes care of the traversal and the `Factory` which maps petal objects to some kind of abstract syntax tree (AST).

For you needs you'll probably just have to subclass the factory and override the methods of interest. The current default implementation is quit simplistic, in particular when mapping associations. It simply maps them to a newly created classes that maintains the connections and adds access methods to the connected classes.

# References

[1] Klasse Objecten, `http://www.klasse.nl/ocl/index.html`. *OCL: the standard constraint language of the UML*, 2001.

[2] Rational, `http://www.rational.com/products/rose/index.jsp`. *Rational Rose*, 2001.

[3] WebGain, `http://www.webgain.com/products/metamata/java_doc.html`. *Java Compiler Compiler*, 2001.

# A Quid identifiers and resolution of object references

The following informations can be found in the FAQ of the Rational Rose web site:

A unique id is simply the current time expressed as the number of seconds elapsed since some point in time in the past, concatenated with some random nuber. The generation algorithm also guarantees that the id will be unique during any given session of Rose and in general, will be unique for all models/units generated on a single machine, assuming the date never gets set backwards. It's not likely that duplicate ids will be generated, however the resolution algorithm makes it even less likely that it would lead to a problem, because:

1. most references are resolved by name first and only if name resolution fails, e.g. the name changed while the referencing item was in an unloaded unit, will it check unique ids.
2. When resolving a unique id reference, the kind of thing being searched for is always used. For example, if the code searches for uid 12345, kind Class, it would never find a package even if there was one that had uid 12345.

How and when Rose needs these UIDs, i.e. how Rose is accessing elements. This is done with the following algorithm:

1. Rose searches after the Qualified Name of the element (i.e. package-hierarchy::element.name
2. When not finding the element, Rose searches within all model elements of the same element type (i.e. it doesn't search the use cases, if it's searching a class) after the element with the same UID.
3. if it doesn't find the element type WITH matching UID, it places the (M) circle sign in the diagram, or places parenthesis around the textual references. (i.e. if you attached a class to an object in a scenario diagram, and then delete the class, the name will stay there, but in paranthesis. If you then recreate the class (which gives it another UID!), it is 'reconnected' to the object)