# The GOODLY Design Language for MOOD Metrics Collection

*Fernando Brito e Abreu*

*Luis Ochoa, Miguel Goulão*

*ISEG / INESC*

## *Abstract*

The GOODLY language can be used to specify the design of applications or libraries built according to the Object Oriented paradigm. It was conceived with the main purpose of facilitating the extraction of MOOD design metrics [Abreu94]. It allows expressing the most relevant design information for that purpose, such as the class structure with corresponding inheritance relations, uses relationships, message exchanges and information hiding. This language, whose syntax and semantics are fully described, is being used in the production of a new generation of the MOODKIT tools that allow the MOOD metrics extraction.

## *1 Introduction*

The first generation of MOODKIT (G1) had a single-tier architecture. Metrics extraction was done in a single step and it was completely independent from formalism to formalism. The lexical-syntactic analyzer for formalism X (for instance C++ source code) had to embed the knowledge about the metrics calculation, instantiated for X case [Abreu95]. For a distinct formalism Y (e.g. Eiffel source code), the lexical-syntactic analyzer had to embed the instantiated knowledge on the metrics calculation for Y [Abreu96b] and so forth. This direct extraction process was then based on a single-tier tool architecture:

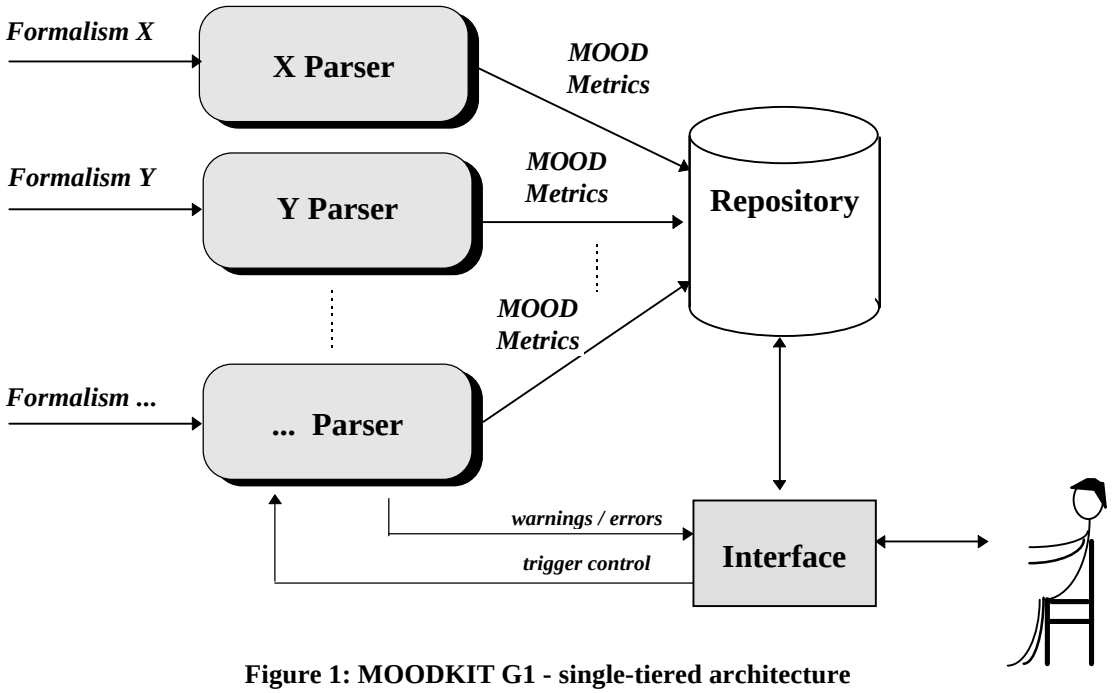


**Figure 1: MOODKIT G1 - single-tiered architecture**

This architecture embodies several drawbacks. The first one arises from the need, and corresponding difficulty, to establish bindings between the concepts used by the specification formalisms and those used in MOOD metrics definition. For each new formalism to be supported, we had to produce a mapping of concepts and embed it in its own lexical-syntactic analyzer. This in turn induced a not less problematic situation: any modification in the MOOD metrics set, such as adding a new metric or simply refining the definition of an existing one, had to be propagated in the code of all analyzers already available.

The growing experience of MOOD usage and corresponding analysis of collected data [Abreu96a] leaded to a step-wise refinement of the set definition. This showed how inflexible the MOODKIT G1 architecture was, to support new metric definitions. The result was a short term unavailability of updated releases for public distribution.

Another MOODKIT G1 shortcoming was the lack of openness. We want to cooperate with other research teams to enlarge the number of formalisms supported by MOODKIT. However, we know how the proliferation of metrics tools leads to distinct values, even when counting such simple things as lines of code. This kind of phenomenon happened within our development team as we produced several versions of MOODKIT G1 (version 1 and 2). Sometimes, a slightly different interpretation of a given concept binding made by two distinct programmers was enough to produce substantially different metrics values. We could easily detect the discrepancies (not so easily its root causes) because we used the same data sets (a collection of libraries) to test new releases.

## 2 MOODKIT and GOODLY

To eliminate the previous problems we conceived a two-tiered architecture for the second-generation (G2) of MOODKIT. A fundamental role is played in this architecture by a design specification language we baptized as GOODLY (a **G**eneric **O**bject **O**riented **D**esign **L**anguage? **Y**es!).

The first tier is made of formalism converters whose function is to produce GOODLY code from specifications originally written in formalisms such as OMT, Fusion, C++, Java, Eiffel or Smalltalk.

The second tier is made of an analyzer of GOODLY code, a repository and a tool interface. The analyzer executes the following operations on the submitted specifications expressed in GOODLY:

- lexical-syntactic verification;
- completeness and referential integrity (traceability) verification through a linking mechanism;
- generation of HT-GOODLY, an hypertext (HTML) version of GOODLY code to allow easy reading (through navigation and context swapping) and understanding;
- MOOD metrics extraction.

The two tiers are then highly decoupled which allows that the first and second tiers may eventually reside in different machines. Since the syntax of GOODLY is formally defined and its semantics is not ambiguous, as we will see in further sections, different teams may cooperate in MOODKIT coverage expansion, by building formalism converters quite independently. The new tool architecture (MOODKIT G2) is then the following:

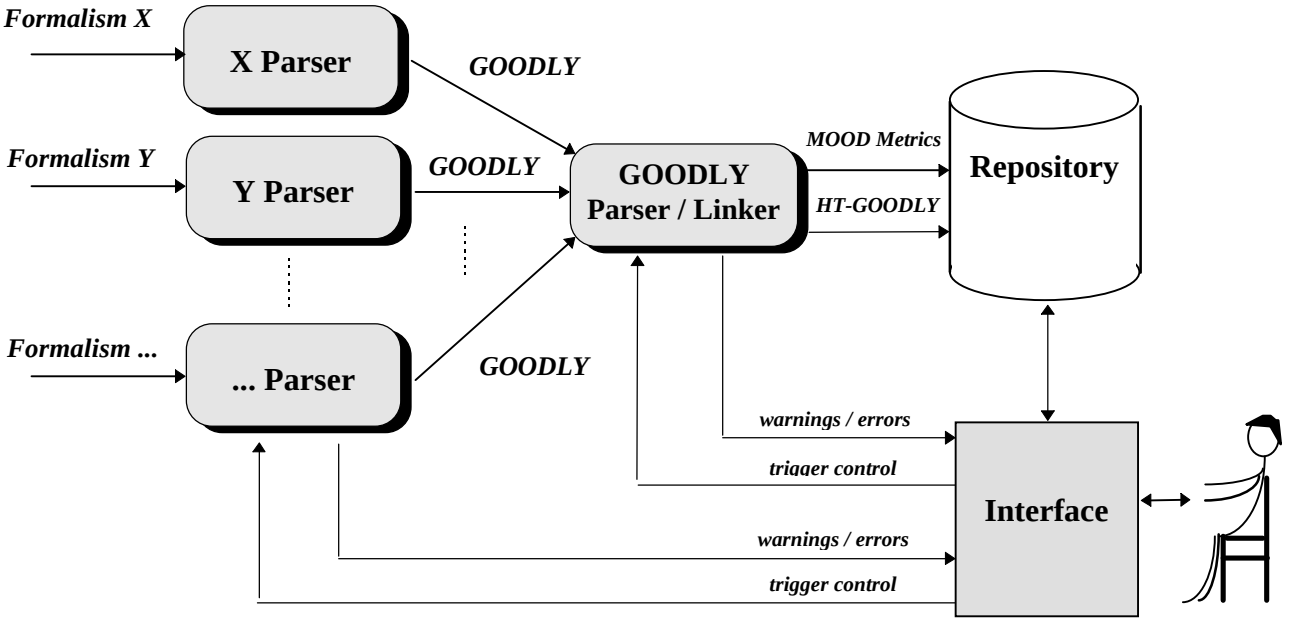


**Figure 2: MOODKIT G2 - two-tiered architecture based on the GOODLY language**

The second tier will be maintained by our team at INESC and made freely available in the Internet. Any evolution in the MOOD set, due to our improved perception enriched by the contributions of all those that will hopefully continue to share with us their comments, will produce changes in the second tier. The formalism converters (first tier), however, are immune to the expected evolution. With available resources, we plan to support (in the second tier) the calculation of other metrics sets such as the one proposed in [Chidamber94].

This new architecture therefore allows proceeding independently in two directions. On one hand, enlarging MOODKIT applicability by adding new formalism converters. On the other hand, facilitating the stepwise refinement and expansion of the MOOD set in an unrestricted way.

## 3 GOODLY generation

It is possible to generate GOODLY specifications either by direct engineering or by reverse engineering. In the former case lies the situation where GOODLY code is produced from OOA&D specifications contained in a CASE tool repository (such as the one of Paradigm Plus, our current case study). The extraction of design information contained in source code written in some OO languages such as C++, Eiffel, Smalltalk, OOPascal, Java, or other is an example of the latter case.

Until now all the research work based on MOOD has only used the reverse engineering approach. The reason is clear: while it is relatively easy (using Internet search engines) to find big volumes of OO source code, such as libraries, the same does not apply to OO analysis and design specifications. Besides, they are usually expressed in some graphical notations, which are usually stored in a non-standardized internal representation, often "hooked" to a certain CASE tool repository. The proliferation of new OO analysis and design formalisms, and the quick evolution of the "older" ones, does not favor the availability of specifications of significant size. Methodologists, for instance, are often more concerned in showing the semantically rich modeling capabilities of their formalisms, rather than in building "real world" examples using them. Hopefully, attempts to standardize some of those formalisms, such as the recently proposed meta-modeling languages UML (Unified Modeling Language) [Booch95] [Booch96] e OML (OPEN Modeling Language) [Firesmith96] are under way.

## 4 The GOODLY Language

### 4.1 General Considerations

The GOODLY design language was conceived to be a common intermediate formalism allowing the extraction of quantitative data - structural software metrics - from the underlying design of systems originally described using other formalisms, either upstream (OO analysis and design models supported in CASE tools), or downstream (programs written in OO programming languages).

In the current stage, this language allows the representation of the structural information considered relevant. GOODLY is not computationally complete: neither algorithmic capabilities, nor control flow structures, are present. Clarity and generality were favored against implementation details. However, for some mechanisms such as the case of visibility and coupling through inheritance, inclusion and message exchange, we wanted it to have sufficient expressive power to allow the representation of most of the corresponding mechanisms in common formalisms such as C++, Java, Smalltalk and Eiffel. To allow the understanding of the design of OO systems, the GOODLY language was enriched (when compared to the previous formalisms) with trace information of such components as classes and modules.

### 4.2 About Types

There are no pre-defined types in GOODLY. All types "imported" from the formalisms in which were written the specifications converted into GOODLY are valid. As in other pure OO languages, such as Smalltalk or Eiffel, the only typing mechanism is the class. All imported types are considered to be classes in GOODLY even if, in hybrid languages such as C++, its internal memory (state) and allowed operations on its instances are not formally encapsulated.

For simplicity sake there is no distinction in GOODLY between access by value and by reference. Pointers (references) for any type are substituted in GOODLY by that same type. As an example a pointer to an instance of class Person is considered in GOODLY as being the instance itself, that is, an object of class Person.

### 4.3 About Style

GOODLY reserved words (terminal symbols) are *case-insensitive* although, for style standardization sake, we generate them always in uppercase. Non-terminals (identifiers of specifications, modules, classes, attributes, operations, etc.) are *case-sensitive* and should retain the letter case they had in its original formalism.

On what concerns indentation, tabs should be used to denote the nesting level embedded in the syntactic structure of this language. Therefore, the sets of instructions between the keywords "XPTO" and END_"XPTO", where "XPTO" stands for terminal symbols such as SPECIFICATION, USES, MODULES, CLASS, STATE, BEHAVIOR,

LOCALS, EMPLOYS, MESSAGES or CALLS, should be skipped one tab to the right of those symbols. The examples included ahead follow this convention.

### 4.4 About the Environment

The validation operated by the GOODLY linker is intended to identify if all components that are part of the system under study are present, with their dependence relationships clearly identified. If there are missing pieces the user is warned with appropriate context-dependent messages.

During compilation and linking, the following global variables are used:

- **CURRENT_SPEC** - identifier of the specification being currently processed;
- **CURRENT_MODULE** - identifier of the module being currently processed;
- **CURRENT_CLASS** - identifier of the class being currently processed.

The following axioms can be applied to these variables:

**∀ <class_spec> , <class_id> : <class_id> ∈ <class_spec> Λ <class_id> = CURRENT_CLASS**

**⇒ <class_spec> ∈ CURRENT_MODULE**

**∀ <module_spec> , <module_id> : <module_id> ∈ <module_spec> Λ <module_id> = CURRENT_MODULE**

**⇒ <module_spec> ∈ CURRENT_SPEC**

## 5 Language Syntax and Semantics

This section includes the complete syntax of the GOODLY language, expressed in BNF notation, along with textual descriptions and examples to allow capturing its semantics.

### 5.1 Specifications and Systems

The structural unit at the highest abstraction level is the **specification**. Its declaration has the following syntax:

*<**specification**>* ::= *<spec_header> <module_list>*
*<spec_header>* ::= *SPECIFICATION <spec_id>*
*TYPE <spec_type>*
*VERSION <version_id>*
*DESCRIPTION <string>*
*OWNER <owner_id>*
*<uses_spec>*
*<modules_spec>*
*<main_spec>*
*END_SPECIFICATION*
*<spec_id>* ::= *<identifier>*
*<spec_type>* ::= *APPLICATION | LIBRARY | ENVIRONMENT | BUILTIN*
*<version_id>* ::= *<string>*
*<owner_id>* ::= *<string>*
*<string>* ::= *“ <text>“*
*<text>* ::= *<char> | <char> <text>*

A software system usually includes the specifications of:

- 1 or more applications (<spec_type> = “**APPLICATION**”);
- 0 or more libraries - sets of reusable components used by the applications (<spec_type> = “**LIBRARY**”);
- 1 development environment being used - described by the set of components that are embedded in it (<spec_type> = “**ENVIRONMENT**”);

- 1 or more graphical or textual specification languages being used, described by the set of embedded components, generally known as pre-defined-types[1] (<spec_type> = "**BUILTIN**").

The specification version can be used for configuration management. The specification header also states who is the owner of the specification (probably who supported its development costs), usually a company, university, some research group or institution, or even some individual. Each specification mentions which others it must use directly [2], so that the origin of all used symbols is known. The corresponding syntax is the following:

*<uses_spec>* *::= USES <used_spec_list> END_USES | <empty>*

*<used_spec_list>* *::= <used_spec_id> | <used_spec_id> , <used_spec_list>*

*<used_spec_id>* *::= <spec_id>*

If one specification does not use any other, then <uses_spec> = <empty>. On the other hand, it does not make sense to indicate that a specification uses itself:

**∀ <specification>, <spec_header>, <spec_id>, <uses_spec>, <used_spec_list> : <spec_header> ∈ <specification> Λ <spec_id> ∈ <spec_header> Λ <uses_spec> ∈ <spec_header> Λ <spec_list> ∈ <uses_spec>**

**⇒ <spec_id> ∉ <used_spec_list>**

One example of a system comprised of several interrelated specifications follows. The meaning of the <modules_spec> and <main_spec> clauses will be detailed in next sections.

```
SPECIFICATION hotel
        TYPE APPLICATION
        VERSION "3.0"
        DESCRIPTION "Hotel management system"
        OWNER "Ecosoft Lda"
        USES
                winmanager, delphi2, oopascal
        END_USES
        MODULES
                front_office, back_office, pos
        END_MODULES
        MAIN
                ...
        END_MAIN
END_SPECIFICATION
```

```
SPECIFICATION delphi2
        TYPE ENVIRONMENT
        VERSION "2.0"
        DESCRIPTION
                "Delphi Development System"
        OWNER "Borland Inc."
        USES
                oopascal
        END_USES
        MODULES
                delphi_features
        END_MODULES
END_SPECIFICATION
```

```
SPECIFICATION winmanager
        TYPE LIBRARY
        VERSION "4.1"
        DESCRIPTION "Window manager library"
        OWNER "Borland Inc."
        USES
                delphi2, oopascal
        END_USES
        MODULES
                win_main, win_aux
        END_MODULES
END_SPECIFICATION
```

```
SPECIFICATION oopascal
        TYPE BUILTIN
        VERSION "2.03"
        DESCRIPTION
              "Built-in OOPascal predefined types"
        OWNER "Borland Inc."
        MODULES
                oopascal_types
        END_MODULES
END_SPECIFICATION
```

[1] - This set was or is being standardized for several OO programming languages.

[2] - By other words, if the specification A uses symbols of specifications B1 and B2, and B1 uses symbols defined in specification C11 and C12, then this "indirect" use in A of symbols defined in C11 and C12 is not enlisted in A.

### 5.2 Main section

A specification can have a start-up section. This section, represented by the <*main_spec*> clause, corresponds to the specification fragment that would be[3] automatically executed whenever the specification is activated. Most specifications of type APPLICATION will have a non-empty clause of this type, while the opposite happens for all other specification types. An exception to this rule is the case where a library with a random number generator needs to be initialized. The initialization procedures for this library (seed generation) should be placed in the corresponding <*main_spec*> clause. The syntax of the <main_spec> clause follows. The syntax of the <*operation_body*> clause will be introduced later.

<*main_spec*> ::= *MAIN* <*operation_body*> <*trace_info*> *END_MAIN* | <*empty*>

### 5.3 Modules

The separated compilation unit in GOODLY is the module. Each module within a specification has a unique identifier. The mechanism for generating identifiers is distinct in direct and reverse engineering.

In direct engineering, that is, when the GOODLY code generation is done from specifications contained in a CASE tool, each module is originated from each defined subsystem. The module names will be those of the corresponding subsystems. If the system to be converted is not partitioned, then only one module will be generated and its name will be the same of the specification.

In reverse engineering, that is, when the GOODLY code generation is done from source code, each module will have the same name as the corresponding source file that is converted. For instance, when the C++ source code file originally designated by “xpto.cpp” is converted, it will originate the module “xpto”.

A specification header generally includes the declaration of which modules it comprises. However, in specifications extracted from analysis and design formalisms supported by CASE tools, the environment may not include a set of components to be reused in the construction of new systems. If that is the case, then <*modules_spec*> = <*empty*>.

**<*modules_spec*>** ::= *MODULES* <*modules_id_list*> *END_MODULES* | <*empty*>
<*modules_id_list*> ::= <*module_id*> | <*module_id*> , <*modules_id_list*>
<*module_id*> ::= <*identifier*>

If <*modules_spec*> is not empty then, besides the declaration of the specificaton header, we have those of the corresponding modules:

<*module_list*> ::= <*module_spec*> | <*module_spec*> <*module_list*> | <*empty*>

The followin assertion then applies:

**∀ <specification>, <spec_header>, <modules_spec> : <spec_header> ∈ <specification> Λ <modules_spec> ∈ <spec_header> Λ <modules_spec> ≠ <empty> Λ <module_list> ∈ <specification> ⇒ <module_list> ≠ <empty>**

Each module in the list is declared with the following syntax:

**<*module_spec*>** ::= *MODULE* <*module_id*>
*DESCRIPTION* <*string*>
*AUTHORS* <*authors_id*>
*PRODUCED* <*timestamp*>
*CONVERTED* <*timestamp*>
<*original_formalism*>
<*class_list*>
*END_MODULE*

---

[3] - Supposing that GOODLY specifications were executable.

The module identifier generation was previously described. A module description and its author(s) are part of each module header. For configuration management purposes, the moments in time when the original fragment was last updated and when it was converted to GOODLY (ex: “23/1/1997_17:53:42”), are also added. The syntax is:

*<authors_id>* *::= <string>*
*<timestamp>* *::= “<date>_<time>“*
*<date>* *::= <day> / <month> / <year>*
*<day>* *::= <number>*
*<month>* *::= <number>*
*<year>* *::= <number>*
*<time>* *::= <hour> : <minute> : <second>*
*<hour>* *::= <number>*
*<minute>* *::= <number>*
*<second>* *::= <number>*

The formalism in which the module was originally written (C++, SmallTalk, Eiffel, Java, OOPascal, OMT, Booch, Fusion, etc) is made explicit by indicating the corresponding specification of “BUILTIN” type:

*<original_formalism>* *::= FORMALISM <builtin_spec_id> | <empty>*
*<builtin_spec_id>* *::= <spec_id>*

The following assertion is applicable here:

**∀ <specification>$_{1,2}$ , <spec_header>$_{1,2}$ , <spec_id>$_{1,2}$ , <spec_type>, <builtin_spec_id> :**

**<spec_header>$_1$ ∈ <specification>$_1$ Λ <spec_id>$_1$ ∈ <spec_header>$_1$ Λ <spec_id>$_1$ = CURRENT_SPEC Λ**

**<specification>$_2$ ≠ <specification>$_1$ Λ <spec_header>$_2$ ∈ <specification>$_2$ Λ <spec_id>$_2$ ∈ <spec_header>$_2$ Λ <builtin_spec_id> = <spec_id>$_2$ Λ <spec_type>$_2$ ∈ <spec_header>$_2$ ⇒ <spec_type>$_2$ = BUILTIN**

In a BUILTIN specification it is not relevant to indicate in which formalism were the modules originally written:

**∀ <specification>, <spec_header>, <spec_type> : <spec_header> ∈ <specification>**

**Λ <spec_type> ∈ <spec_header> Λ <spec_type> = BUILTIN ⇒ <original_formalism> = <empty>**

As an example we include a BUILTIN specification with a single module:

```
SPECIFICATION xpto
        TYPE BUILTIN
        VERSION “1”
        DESCRIPTION
                “Hypothetical formalism supporting
                a given set of basic types”
        OWNER “MOOD Project”
        MODULES
                xpto_types
        END_MODULES
END_ SPECIFICATION
```

```
MODULE xpto_types
        DESCRIPTION
                “Basic types in the xpto formalism”
        AUTHORS “Fernando Brito e Abreu et al.”
        PRODUCED “18/7/96_10:30:15”
        CONVERTED “20/8/96_23:12:57”

        CLASS Char END_CLASS
        CLASS String END_CLASS
        CLASS Integer END_CLASS
        CLASS Real END_CLASS
        CLASS Boolean END_CLASS
        CLASS Date END_CLASS
        CLASS Time END_CLASS
END_MODULE { xpto_types }
```

### 5.4 Specification and Module Persistance

Each specification (header and corresponding modules) will be kept in a separate subdirectory whose name will be the same as that of the specification it will contain, with no extension. This subdirectory will be located under the "**specs**" directory which, in turn, is located under the directory identified by the environment variable named GOODLYDIR.

A specification header resides on a separated file and identifies which modules belong to the specification. This header is used by the GOODLY link editor to find symbols referenced in the modules. A file that contains the header of one specification will have a descriptor equal to the specification name, plus the ***spc*** extension. For instance a specification named "*salaries*" would be kept in the file "*salaries.spc*".

Each module is also stored in a distinct file whose name is that of the module it contains plus the **gly** extension.

Supposing that the environment variable GOODLYDIR has the value "/usr/etc/goodly", then the persistent storage of the hotel and other related specifications introduced beforehand, would be[4]:

***/usr/etc/goodly/specs/hotel/hotel.spc***
***/usr/etc/goodly/specs/hotel/front_office.gly***
***/usr/etc/goodly/specs/hotel/back_office.gly***
***/usr/etc/goodly/specs/hotel/pos.gly***

***/usr/etc/goodly/specs/lists/lists.spc***
***/usr/etc/goodly/specs/lists/listpack.gly***

***/usr/etc/goodly/specs/winmanager/winmanager.spc***
***/usr/etc/goodly/specs/winmanager/win_main.gly***
***/usr/etc/goodly/specs/winmanager/win_aux.gly***

***/usr/etc/goodly/specs/delphi2/delphi2.spc***
***/usr/etc/goodly/specs/delphi2/delphi_features.gly***

***/usr/etc/goodly/specs/oopascal/oopascal.spc***
***/usr/etc/goodly/specs/oopascal/oopascal_types.gly***

### 5.5 Classes and their Parameters

The basic component of a module is the **class**. A module is no more than a set of classes which were physically aggregated on the basis of a given modularization criterion. That criterion should be made explicit, for instance in the module description clause. Each class must have a unique identifier within each specification.

| | |
|---|---|
| *<class_list>* | *::= <class_spec> \| <class_spec> <class_list>* |
| ***<class_spec>*** | *:: = CLASS <identifier> <class_parameters> <inheritance_spec>* |
| | *<state_spec>* |
| | *<behavior_spec>* |
| | *<trace_info>* |
| | *END_CLASS* |
| *<class_parameters>* | *::= [ <parameter_id_list> ] \| <empty>* |
| *<parameter_id_list>* | *::= <parameter_id> \| <parameter_id>* **,** *<parameter_id_list>* |
| *<parameter_id>* | *::= <class_id>* |
| *<class_id>* | *::= <identifier> <class_parameters> \| UNKNOWN* |

The UNKNOWN keyword is used when it is not possible to determine the appropriate class identification at linking time. This situation does not arise in the class declaration but rather in the references that can be made to classes in other clauses.

A class with no parameters has an empty *<class_parameters>* clause. The class parameterization supported here is a mechanism similar to those existing in languages such as C++ and Eiffel that allows building generic classes ("*templates*"). An example follows:

```
CLASS Queue [Something]
        ...
END_CLASS
```

[4] - On an operating system supporting long names; otherwise file identifiers can be truncated.

Based in this class we could get parameterized situations such as the following:

```
waiting_list: Queue [People];
doctor_booking: Queue [Patient];
loan_list: Queue [Request];
```

### 5.6 Inheritance

The GOODLY language supports the specification of both single and multiple inheritance. It is not made any distinction between interface inheritance that implies an implicit type conversion (*sub-typing*) and implementation inheritance (ex: private inheritance in C++). As an example:

```
CLASS Airplane                          {base class; no inheritance}
        ...
END_CLASS

CLASS Fighter ISA Airplane              {single inheritance}
        ...
END_CLASS

CLASS Seaplane ISA Airplane, Ship       {multiple inheritance}
        ...
END_CLASS
```

Formally we have:

<*inheritance_spec*> ::= *ISA* <*class_id_list*> | <*empty*>
<*class_id_list*> ::= <*class_id*> | <*class_id*> **,** <*class_id_list*>

There is no inheritance clause for a root class (class with any ancestors). Then we have <inheritance_spec> = <empty>

### 5.7 Traceability

When the link editing takes place in GOODLY, trace information is automatically inserted in the source code with the syntax expressed in clause <trace_info>. That information consists on the identification of the places where the classes locally referenced [5] are defined. If the code to be linked already has traceability clauses, because it has been previously submitted to the link editor, these will be discarded and rewritten to guarantee a true conformity with the version whose modules were effectively linked.

<*trace_info*> ::= *TRACE* <*traced_class_list*> *END_TRACE* | <*empty*>
<*traced_class_list*> ::= <*traced_class_id*> | <*traced_class_id*> **;** <*traced_class_list*>

The traceability clause will only be inserted if it contains at least one entry; otherwise we get <trace_info> = <empty>.

<*traced_class_id*> ::= <*class_id*> <*origin_module*>
<*origin_module*> ::= *UNDEFINED* | *DEFINED_IN* <*module_id*> <*origin_spec*>
<*origin_spec*> ::= *OF* <*spec_id*> | <*empty*>

The profusion of traceability information depends solely on the context, as follows:

- if the class referenced is defined in a **module belonging to a BUILTIN specification**, then no traceability information is added, that is, <*traced_class_id*> = <*empty*>. This avoids the proliferation of "unnecessary" information (pre-defined types are usually well known)*;*

[5] - the locality here corresponds to the clause to which the non-terminal symbol <*trace_info*> is associated;

- if the class referenced is defined in the **same module** then, given the proximity, no traceability information is added, that is, *<traced_class_id> = <empty>;*
- if the class referenced is defined in the **same specification but not in the same module,** then it is only added information about the corresponding module. Thus we have *<origin_spec> = <empty>;*
- if the class referenced is defined in a **module of other specification,** then it is added information about both of them. Thus we have *<origin_spec> ≠ <empty>;*
- if the class referenced was **not found in any module of involved specifications**, then it is added information stating that situation (<origin_module> = UNDEFINED).

### *5.8 Object State*

The specification of a class has two parts: the **state** and the **behavior**. The state of each object (class instance) depends on the value of the attributes it contains and which are defined in the corresponding class. Each attribute [6] has an identifier, a type (class) and a scope. Where applicable, its type definition will be enriched with traceability information generated in the previously described fashion.

Some OO programming languages such as Smalltalk make a distinction between class state (class variables) and object state (instance variables). Since GOODLY is not an executable formalism, this distinction is not made and both cases are included in the <state_spec> clause:

***<state_spec>*** *::= STATE <attribute_list> END_STATE | <empty>*

If a class has no attributes then the state clause is omitted. Then we get <state_spec> = <empty>

*<attribute_list>* *::= <attribute> | <attribute>* ***;*** *<attribute_list>*
*<attribute>* *::= <unscoped_attribute> <scope_spec>*
*<unscoped_attribute>* *::= <attribute_id> : <class_id_list>* [7]
*<attribute_id>* *::= <identifier>*

### *5.9 Scope of Attributes and Operations*

The scope or range of an attribute or operation (clause *<scope_spec>*) can be defined in a very detailed fashion. The scope is characterized by the visibility that components (classes) have on the attribute or operation. Invisibility implies inability to use. The following scope options can be used:

- **PRIVATE** scope - only the class where the attribute or operation is defined can "see" it; this scope is not cumulative with any other;
- **PROTECTED** scope - the class where the attribute or operation is defined, and its descendants, can "see" it;
- Discriminated class scope - the class referenced by ***<class_id>*** can "see" the attribute or operation; the referenced class must be defined in the same specification, although it can be declared in another module;
- Discriminated class hierarchy scope - the class referenced by ***<class_id>*** and its descendants can "see" the attribute or operation; the referenced class is supposed to be defined in the same specification, although it can be declared in another module. The * suffix represents the inheritance hierarchy starting in the class where it is appended; therefore ***<class_id>*** .* stands for the class referenced by ***<class_id>*** and all its descendants;
- **MODULE** scope - all classes in the module that contains the class where the attribute or operation is defined can "see" it;
- Specification (**SPEC**) scope- all classes in the specification that contains the class where the attribute or operation is defined can "see" it;

---

[6] - in other formalisms these are designated by instance variables or data members;

[7] - In most cases <class_id_list> will only have one element. In some cases (ex: Smalltalk) however, the same object identifier can be associated with instances of distinct classes.

- **PUBLIC** scope - all classes can "see" the attribute or operation, regardless of being defined in the same or other specification.

*<scope_spec>* *::= SCOPE <scope_list>*
*<scope_list>* *::= <scope_id> | <scope_id> , <scope_list>*
*<scope_id>* *::= PRIVATE | PROTECTED | MODULE | SPEC | PUBLIC | <class_id> | <class_id>.**

The scope of an attribute or operation always includes the own class where it is defined. Therefore it does not make sense to include it explicitly [8] in the scope clause. Formally we have the following assertion:

**∀ <class_id>, <scope_id> : <class_id> ∈ <scope_list> ⇒ <class_id> ≠ CURRENT_CLASS**

The resulting scope, when a list of scope options is used, is the union of the class sets corresponding to each list element when considered individually. Because some of the scopes are more ample than others, their simultaneous use can become redundant. That will happen when a scope embraces some other. The PUBLIC scope, for instance, embraces any other and so it can be considered as the absorbing [9] element in a scope clause. The PRIVATE scope is more restrictive than any other is and so it can be considered as the neutral [10] element in a scope clause. The SPEC scope embraces the MODULE scope and the discriminated class scope. The MODULE scope can embrace a discriminated class if that is defined within the current module. It can further embrace the PRIVATE and discriminated class hierarchy scopes if the corresponding descendants are also contained in the current module.

The following table shows the resulting scope of the union, two by two, of named scope options. It can be used recursively to determine the resulting scope of any list of scope options. The OWN acronym represents the current class, that is, the one where is defined the attribute or operation we are concerned with. For simplification purposes classes C1 and C2 and all their descendants were considered to belong to the current module.

| | **PRIVATE** | **PROTECTED** | **CLASS C2** | **CLASS C2.*** | **MODULE** | **SPEC** | **PUBLIC** |
|---|---|---|---|---|---|---|---|
| **PRIVATE** | OWN | OWN.* | {C2, OWN} | {C2.*, OWN} | MODULE | SPEC | PUBLIC |
| **PROTECTED** | OWN.* | OWN.* | {C2, OWN.*} | {C2.*, OWN.*} | MODULE | SPEC | PUBLIC |
| **CLASS C1** | {C1, OWN} | {C1, OWN.*} | {C1, C2, OWN} | {C1, C2.*, OWN} | MODULE | SPEC | PUBLIC |
| **CLASS C1.*** | {C1.*, OWN} | {C1.*, OWN} | {C1.*, C2, OWN} | {C1.*, C2.*, OWN} | MODULE | SPEC | PUBLIC |
| **MODULE** | MODULE | MODULE | MODULE | MODULE | MODULE | SPEC | PUBLIC |
| **SPEC** | SPEC | SPEC | SPEC | SPEC | SPEC | SPEC | PUBLIC |
| **PUBLIC** | PUBLIC | PUBLIC | PUBLIC | PUBLIC | PUBLIC | PUBLIC | PUBLIC |

**Table 1: Resulting scope combinations**

## *5.10 Object Behavior*

### *5.10.1 Operation Interface*

The behavior of all instances of the same class (objects) is common and is defined in the class to which they belong as a set of **operations** [11]. Each operation has an interface and a body or implementation. The **operation interface**

---

[8] - Although its reference is implicit in the PRIVATE and PROTECTED scope options.

[9] - In the sense that the other scopes that could be included in the same scope specification would be rendered irrelevant.

[10] - In the sense that any other scope that could be included in the same scope specification would include the private scope, thus rendering its declaration irrelevant. In other words, it only makes sense to declare a private scope when that is the only one applicable.

[11] - In other formalisms these are often designated by methods or function members.

includes its identifier, the formal parameter list and corresponding type(s), the returning type, its scope and traceability information. The corresponding syntax is the following:

***<behavior_spec>*** *::= BEHAVIOR <operations_list> END_BEHAVIOR | <empty>*
*<operations_list>* *::= <operation> | <operation> <operation_list>*
*<operation>* *::= OPERATION <operation_interface>*
*<operation_body>*
*END_OPERATION*

*<operation_interface>* *::= <operation_id> (<parameters>) <return_spec> <scope_spec>*
*<operation_id>* *::= <identifier> | <operator>*
*<operator>* *::= '+' | '-' | '++' | '--' | '=' | '<>' | '!=' | '>' | '>=' | '<' | '<=' | '&&' | '||'*
*<parameters>* *::= <parameter_list> | <empty>*
*<parameter_list>* *::= <parameter> | <parameter>* ***;*** *<parameter_list>*
*<parameter>* *::= <unscoped_attribute>* [12]
*<return_spec>* *::= : <class_id> | <empty>*

If a class has no defined operations, then the behavior clause is omitted, that is <behavior_spec> = <empty>. The body of an operation is also optional. That is often the case in third party libraries where only the behavior interface is available.

### *5.10.2 Operation Body*

The implementation or **operation body** includes the definition of its local variables, the attributes it employs, the messages that this operation sends to instances of this or of other classes and the calls that it makes to services of entities other than class instances. The corresponding syntax is:

*<operation_body>* *::= <locals_spec> <employs_spec> <messages_spec> <calls_spec> | <empty>*

When compared with a computationally complete language, the GOODLY language does not include the entire algorithmic component (instruction sequence that produce a given result) but only the information on collaborations and responsibilities of classes among themselves. In the body of one operation, the references made to attributes and operations declared in other classes are prefixed with their respective class name. Referenced attributes and operations declared in the same class do not carry the prefix.

A given operation can manipulate several variable types:

- temporary variables locally defined (usable only within its scope);
- attributes belonging to the current class;
- attributes belonging to other classes;
- other variables who are not associated with any class such as environment variables, global variables, etc.

The variables of the first type are described in the next section.

#### 5.10.2.1 Local Variables

The local variables are similar to the attributes that define the state, but with no scope information. Their declaration syntax is the following:

*<locals_spec>* *::= LOCALS <local_attribute_list> END_LOCALS | <empty>*

---

[12] - In most programming languages when a parameter is passed by value, the actual parameter list can include a variable, constant, value returned by a function or any kind of compatible type expression. In GOODLY the actual parameter lists (as will be seen in the *message* and *call* specifications ahead) will, at most, contain the identifier and type of the corresponding formal parameter variables. By other words, there is no distinction here between formal and actual parameter lists.

*<local_attribute_list> ::= <local_attribute> | <local_attribute>* ***;*** *<local_attribute_list>*

*<local_attribute> ::= <unscoped_attribute>*

If an operation has no local variables then the <locals_spec> clause can be omitted, that is <locals_spec> = <empty>.

#### 5.10.2.2 Direct Access to Object State

An operation can access the instance attributes of the same or other classes. This kind of coupling is described in the following clauses:

*<employs_spec> ::= EMPLOYS <employed_attribute_list> END_EMPLOYS | <empty>*

*<employed_attribute_list>::= <employed_attribute> | <employed_attribute>* ***;*** *<employed_attribute_list>*

*<employed_atribute> ::= <attribute_id> | <class_id>.<attribute_id>*

As previously, the prefix *<class_id>* is applied to the *<employed_attribute>* when this does not correspond to an attribute defined in the clause *<state_spec>* of the same class.

The *<employs_spec>* clause only refers to direct[13] manipulations of attributes that describe the state of objects, either in the same or other class.

#### 5.10.2.3 Messages

To accomplish a given task an operation frequently uses the services of other operations. By other words, other classes (or operations defined in the same class) can cooperate in the prosecution of a given operation. Their co-operation is obtained by invoking its services through message sending. The description of which messages an operation sends is included in the following clause:

*<messages_spec> ::= MESSAGES <message_list> END_MESSAGES | <empty>*

*<message_list> ::= <message> | <message>* ***;*** *<message_list>*

*<message> ::= <message_id> | <message_id> (<parameters>)<return_spec>*

If an operation does not send messages, or if those are unknown because only interface information is available, then <messages_spec> = <empty>. If a message corresponds to an operation that is unambiguously determined only by its identifier, then, in the previous clause, only that identifier is used. However some formalisms, such as C++, allow that several operations with the same identifier (but with distinct formal parameter lists) can be defined in the same class. In this case the corresponding messages must indicate, extensively, the formal parameter lists to withdraw the ambiguity, as shown in the next clause:

*<message_id> ::= <operation_id> | <class_id>.<operation_id>*

As it occurs in the *<employed_attribute>* clause, the *<class_id>* prefix is added to the *<message>* when this does not correspond to an operation defined in the *<behavior_spec>* of the same class; by other words, when the message recipient is an instance of other class.

#### 5.10.2.4 Service Calls

To accomplish its objective an operation can invoke services from entities other than class instances as, for instance, a window manager, operating system, network manager or other. These calls are represented in the following clause:

*<calls_spec> ::= CALLS <call_list> END_CALLS | <empty>*

*<call_list> ::= <call> | <call>* ***;*** *<call_list>*

*<call> ::= <call_id> (<parameters>) <return_spec>*

*<call_id> ::= <identifier>*

---

[13] - Not through message exchanges.

Such as in the case of messages, if an operation does not use calls or if those are unknown because only interface information is available, then <calls_spec> = <empty>.

### 5.11 Identifiers and Comments

This section includes some definitions of symbols used in previous clauses, which were still missing.

| | |
|---|---|
| *<empty>* | *::= ‘’* |
| *<identifier>* | *::= <alphanumeric> \| <alphanumeric> <identifier>* |
| *<alphanumeric>* | *::= <letter> \| <digit> \| ‘_’ \| ‘:’* |
| *<letter>* | *::= ‘A’..’Z’ \| ‘a’..‘z’* |
| *<char>* | *::= <alphanumeric> \| <punctuation>* |
| *<punctuation>* | *::= ‘,’ \| ‘;’ \| ‘.’ \| ‘?’ \| ‘!’ \| ‘’’ \| ‘“‘ \| ‘/’ \| ‘-’ \| ‘_’ \| ‘{‘ \| ‘}’ \| ‘#’ \| ...* |

Comments can appear in any point of a specification, delimited by brackets.

| | |
|---|---|
| *<comments>* | *::= { <text> }* |

## 6 A short example

To illustrate the syntax and semantics of the GOODLY language we include hereafter the following specification of a hypothetical application in the car races domain:

```
SPECIFICATION  An_example
        TYPE APPLICATION
        VERSION “2.1”
        DESCRIPTION “A rally competition”
        OWNER “MOOD Project”
        USES
                xpto
        END_USES
        MODULES
                Race
        END_MODULES

        MAIN
                LOCALS
                        mikkola: Pilot;
                        chuck: Mechanic;
                END_LOCALS

                MESSAGES
                        Pilot.creates;
                        Mechanic.creates;
                        Pilot.qualification;
                        Pilot.talks_to      {Mechanic};
                        Mechanic.repairs;
                        Pilot.qualification;
                        Pilot.talks_to      {Mechanic};
                        Mechanic.repairs;
                        Pilot.qualification;
                        Pilot.destroys;
                        Mechanic.destroys
                END_MESSAGES

                CALLS ... END_CALLS
        END_MAIN
END_ SPECIFICATION
```

```
MODULE Race
        DESCRIPTION "Example of a module of an application in the race car world"
        AUTHORS "Fernando Brito e Abreu et al."
        PRODUCED "18/7/96_12:00:03"
        CONVERTED "20/8/96_15:34:45"
        FORMALISM xpto

        CLASS Document[Objective] END_CLASS

        CLASS Sport END_CLASS

        CLASS Bank END_CLASS

        CLASS Person
                STATE
                        name: String SCOPE PRIVATE;
                        birth_date: Date SCOPE PRIVATE;
                        weight: Integer SCOPE PRIVATE;
                        address: String SCOPE PRIVATE;
                        bankcard: Document[Bank] SCOPE PRIVATE
                END_STATE

                BEHAVIOR
                        OPERATION creates( ) SCOPE PUBLIC END_OPERATION
                        OPERATION destroys( ) SCOPE PUBLIC END_OPERATION
                END_BEHAVIOR
        END_CLASS { Person }

        CLASS Pilot ISA Person
                STATE
                        driving_permit: Document[Sport] SCOPE PRIVATE;
                        helmet: Protection SCOPE PROTECTED
                END_STATE

                BEHAVIOR
                        OPERATION qualification(start: Time; from: String; to: String) SCOPE PUBLIC
                                MESSAGES
                                        Protection.creates;
                                        Protection.put_on;
                                        Document[Sport].shows;
                                        RaceCar.close_window;
                                        RaceCar.starts;
                                        drives;
                                        RaceCar.stops;
                                        RaceCar.open_window;
                                        Document[Sport].shows;
                                        Protection.take_off;
                                        Protection.destroys
                                END_MESSAGES
                        END_OPERATION

                        OPERATION drives(vehicle : RaceCar): Boolean SCOPE PUBLIC
                                MESSAGES
                                        Co-pilot.read_notes;
                                        RaceCar.accelerates;
                                        Co-pilot.read_notes;
                                        RaceCar.breaks;
                                        RaceCar.turns
                                END_MESSAGES
                        END_OPERATION
```

```
                OPERATION talks_to(whom: Person): Boolean SCOPE Co-pilot, Mechanic
                END_OPERATION
        END_BEHAVIOR
END_CLASS {Pilot}

CLASS Co-pilot ISA Person
        BEHAVIOR
                OPERATION read_notes( ) SCOPE PUBLIC END_OPERATION
        END_BEHAVIOR
END_CLASS { Co-pilot }

CLASS Mechanic ISA Person
        BEHAVIOR
                OPERATION repairs (vehicle : RaceCar) SCOPE PUBLIC END_OPERATION
        END_BEHAVIOR
END_CLASS { Mechanic }

CLASS Artifact
        STATE
                brand: String SCOPE PUBLIC;
                model: String SCOPE PROTECTED;
                production_date: Date SCOPE PRIVATE
        END_STATE

        BEHAVIOR
                OPERATION creates( ) SCOPE PUBLIC END_OPERATION;
                OPERATION destroys( ) SCOPE PUBLIC END_OPERATION
        END_BEHAVIOR
END_CLASS { Artifact }

CLASS Protection ISA Artifact
        BEHAVIOR
                OPERATION put_on( ): Boolean SCOPE PUBLIC END_OPERATION
                OPERATION take_off( ): Boolean SCOPE PUBLIC END_OPERATION
        END_BEHAVIOR
END_CLASS { Protection }


CLASS RaceCar ISA Artifact
        STATE
                zero_to_hundred: Time SCOPE PUBLIC;
                horse_power: Integer SCOPE PUBLIC;
                oil_level: Real SCOPE Mechanic.*, Pilot, Co-Pilot;
                actual_speed: Integer SCOPE PROTECTED, Pilot, Co-Pilot
        END_STATE
        BEHAVIOR
                OPERATION starts( ): Boolean SCOPE PUBLIC END_OPERATION
                OPERATION accelerates (duration: Time; acceleration: Integer) SCOPE PUBLIC
                END_OPERATION
                OPERATION breaks(duration: Time; desacceleration: Integer) SCOPE PUBLIC
                END_OPERATION
                OPERATION turns(angle: Integer ) SCOPE PUBLIC END_OPERATION
                OPERATION stops( distance: Integer ): Boolean SCOPE PUBLIC
                END_OPERATION
                OPERATION open_window( ): Boolean SCOPE PUBLIC END_OPERATION
                OPERATION close_window( ): Boolean SCOPE PUBLIC END_OPERATION
        END_BEHAVIOR
END_CLASS { RaceCar }

END_MODULE { Race }
```

## 7 Final considerations

This paper described the architectural evolution of MOODKIT, a set of tools to collect the MOOD metrics, along with its rationale. The GOODLY design language plays a fundamental role in this effort. A full description of the language syntax and semantics was included. Some examples written in this language were also included.

The foundations for the cooperation with other research teams in the OO metrics collection arena are now laid. We hope to strengthen the joint efforts already in place and to launch new ones.

To browse HT-GOODLY examples and to access available MOODKIT G1[14] versions and a database of collected metrics plus the original code from where they were extracted, and to get papers produced within the MOOD project, visit our web site (**http://albertina.inesc.pt/ftp/pub/esw/mood**) or ftp to **albertina.inesc.pt** (user: anonymous; password: your email address; directory: pub/esw/mood).

## 8 References


[Abreu94] Abreu, F. Brito and Carapuça R., "Object-Oriented Software Engineering: Measuring and Controlling the Development Process", Proceedings of the 4th International Conference on Software Quality, ASQC, McLean, VA, USA, October 1994.

[Abreu95] Abreu, F. Brito; Goulão, Miguel and Esteves, Rita, "Toward the Design Quality Evaluation of Object-Oriented Software Systems". Proceedings of the 5th International Conference on Software Quality, Austin, Texas, USA, October 1995.

[Abreu96a] Abreu, F. Brito and Melo, Walcelio, "Evaluating the Impact of Object-Oriented Design on Software Quality". Proceedings of the Third International Software Metrics Symposium, Berlin, March 1996.

[Abreu96b] Abreu, F. Brito; Esteves, Rita and Goulão, Miguel, "The Design of Eiffel Programs: Quantitative Evaluation Using the MOOD Metrics", Proceedings of TOOLS USA'96 (Technology of Object Oriented Languages and Systems), Santa Barbara, California, USA, August 96.

[Booch95] Booch, Grady; Rumbaugh, James, "Unified Method for Object-Oriented Development - Documentation Set", Version 0.8, Rational Software Corporation, October 1995.

[Booch96] Booch, Grady; Rumbaugh, James; Jacobson, Ivar, "Unified Modeling Language - Documentation Set", Version 0.9, Rational Software Corporation, July 1996.

[Chidamber94] Chidamber S. and Kemerer C., "A metrics suite for object oriented design", *IEEE Transactions on Software Engineering*, vol. 20, n. 6, pp. 476-493, June 1994.

[Firesmith96] Firesmith, Donald; Henderson-Sellers, Brian; Graham, Ian, "OPEN Modeling Language (OML) - Core Notation Specification", Version 0.1, OPEN Consortium, July 1996.


[14] - G2 versions will also be available in the near future!