ELSEVIER

The Journal of Systems and Software 50 (2000) 57–64

The Journal of Systems and Software

www.elsevier.com/locate/jss

Module interconnection features in object-oriented development tools

Sergio Eduardo Rodrigues de Carvalho *, Julio Cesar Sampaio do Prado Leite

Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro, R. Marquês de São Vincente 225, 22453-900 Rio de Janeiro, Brazil

Received 13 January 1998; received in revised form 19 April 1998; accepted 9 June 1998

Abstract

The black-box reuse of library classes in the construction of an object-oriented (OO) application is difficult: the principle of information hiding may be violated if classes must know their partners, and code must be typically rewritten. One of the possible ways to increase class decoupling and thus reuse is to employ interconnection languages to describe OO architectures. In this paper we present a decoupling paradigm for individual classes or class collections that facilitates reuse, introducing interconnection features at the design and programming levels. We give examples in a new, second generation OO development system, using asynchronous messages sent to generically identified receivers. © 2000 Elsevier Science Inc. All rights reserved.

1. Introduction

The object-oriented (OO) technology provides a welcoming framework for software reuse, but as it has been correctly pointed out (Griss, 1994), its use does not guarantee reuse. In order to attain reuse it is necessary to plan for reuse, independently of which technology is in place. It is also well known that implementing the reuse state of mind is not just a technical problem. In this article we will focus on the technological side of reuse. We will show how features of a new OO development system, 2GOOD (Second Generation Object-Oriented Development), help define reusable components and architectures.

Central to the design of systems, which, in any development paradigm, are composed of parts, is the issue of how we describe the interconnection of those parts. Module Interconnection Languages (MIL's) (Prieto-Diaz and Neighbors, 1986) have been proposed as a means of describing the structure of conventional software systems, and can be seen as important artifacts to minimize "glue code" – code that needs to be written, when creating a new application, to allow modules being put together to interface with each other (Neighbors, 1994). Although MIL's may help avoiding glue code, Biggerstaff points out that typically MIL's do not deal with vertical and horizontal composition, [1] and that the MIL/implementation language gap is not negligible (Biggerstaff, 1994). LIL (Goguen, 1986), LILEANA (Tracz, 1993) and OOMIL (Hall and Weedon, 1993) are interconnection languages that address vertical and horizontal composition (Goguen, 1996), but they need programming systems that implement in a proper manner the notion of composition that they embody.

More recently, architectural description languages are being studied (Shaw and Garlan, 1996) as a means to fulfill the role of MIL's in the development of large software systems. One of the main issues in this endeavor is the representation of module connectors, which should be considered first class entities in that development. This idea is also embraced by (Goguen, 1996), who advocates the use of parameterized programming for describing software architectures.

Perhaps one of the problems with the reuse of OO systems is the under-utilization of the composition mechanism. For some time the literature over-emphasized the role of inheritance as the way of getting reuse, and less attention was

* Corresponding author.

E-mail addresses: sergio@inf.puc-rio.br (S.E.R. Carvalho), julio@inf.puc-rio.br (J.C.S. do Prado Leite).

[1] A composition is a record-like structure, as in conventional languages.Vertical composition refers to composition using lower, less abstract layers, whereas horizontal composition refers to composition on the same layer (see Goguen ,1996, for an example).

PII: S0164-1212(99)00076-X

given to composition. Current work in frameworks (CACM, 1997) and design patterns (Gamma et al., 1995) tends to assert a renewed importance to composition. However, composition is not problem-free, as the library scaling problem – the reuse difficulty that occurs when libraries contain a large number of classes (Biggerstaff, 1994) – and the related problem of glue code seem to indicate.

Composition problems are being addressed in development systems being researched at the University of Texas at Austin (GenVoca) (Batory et al., 1994), and at the University of Berne (Nierstrasz and Tsichritzis, 1995). Both are essentially based on object technology, and aim at the development of plug-and-play components for reusable architectures. The 2GOOD features we are illustrating in this paper could be useful to both approaches.

In the OO paradigm, classes or collections of classes are the generally accepted units of reuse, the "modules" to be interconnected. This is no great loss of generality, since for example the functionality of entire collections of classes, or subsystems, can be presented to the outside world via a single class, using for example the Facade or Mediator patterns [2] (Gamma et al., 1995); also, incomplete classes, encapsulating only data or algorithms, can be defined to wrap up external environments.

The nature of classes is such that they already include features usually found in conventional MIL's: composition is one of them. A class defines the structure of its objects, and this is a composition of objects, modeled by classes obviously being reused.

In this paper we show how we can achieve a higher degree of reuse, by designing classes and class collections independently of the context where they are to be used. We argue that we can profit from mechanisms guaranteeing, for example, that the communication between components and the containing whole is "impersonal": component objects should be able to communicate the results of their efforts without explicitly naming the whole object that contains them. With this feature we can achieve context independent reusability (black-box classes) and further client–server and layered architectures. Furthermore, we claim that the expression of this impersonal communication should be possible at various development stages, to avoid unduly restricting the creativity of designers, and allowing design for reuse. Modern OO development systems should also bridge the design/implementation gap, offering automatic transformations between phases, minimizing discontinuities and providing traceability.

To attain these results we need a second generation of OO development systems, of which 2GOOD is an example. We achieve black-box reusability in 2GOOD designing classes modeling objects and object systems that can send asynchronous messages to generically identified receivers, as described below. Sections 2 and 3 shortly describe 2GOOD, with emphasis on its message passing mechanisms and interconnection features. In Section 4 we show how message passing in 2GOOD supports OO reuse, with the aid of a known example. In Sections 5 and 6 we discuss alternatives to the proposed mechanism, and show its applicability in the use of applications as components. Section 7 discusses the results obtained, and presents future research.

2. The 2GOOD development system

The 2GOOD development system (Carvalho et al., 1996a) includes a newly developed CASE tool allowing the construction of typical static and dynamic OO system views, such as class relationships, object communication, state and scenario diagrams. The system does not impose specific methodologies in constructing applications or class libraries. Using this visual design/programming tool, which continuously verifies consistency, it is possible to automatically obtain code corresponding to the current development stage, first in a very high-level design description language (DDL) (Carvalho et al., 1996b), and then in C++. A DDL specification, an automatically obtained textual representation of system diagrams, is a collection of classes, newly defined in the system or reused from libraries. To enhance reuse, there is no global referencing environment.

Classes in 2GOOD are the meeting grounds for all properties associated with the objects they model. Among these properties we find a data structure describing the composition of class objects, a section defining the message sending behavior of objects, and a collection of operations describing how class objects respond to stimuli. These class properties will be directly related to MIL features in what follows. We can select different semantics to express class operations. In this paper we focus on asynchronous messages, which do not block the sender, so sender and receiver can concurrently operate on different threads.

[2] Abstract architectures, involving a small number of classes, frequently used in OO development.

3. Interconnection features in 2GOOD

MIL's describe high-level designs, centering the attention on the interfaces between parts and their composition in a given system. In MIL's all resources are provided by "modules". Syntax primitives describe the flow of resources and are usually designated by names such as *import*, *export*, *requires*, *provides*, *needs*, *contains*, *uses*, *consists-of* and others. In general, OO design approaches do not explore the composition aspect of design in the line followed by MIL's. Hall's proposal (Hall and Weedon, 1993) is one of the few considering objects in the design of a MIL.

In 2GOOD the message section and the declaration of operations we find in classes are representatives of the concepts of *export* and *import* clauses used in MIL's. In the message section we define the asynchronous messages that can be sent by class objects to their clients. This constitutes the *export* clause of the class. The operations deal with requests sent by client objects. This constitutes the *import* section of the class.

Asynchronous message handlers are the operations needed to process asynchronous messages received by an object. They are specially convenient in the design of black-box, reusable classes, when combined with object relationships automatically discovered by the system. In this paper we focus on one such relationship: PARENT/child (PARENT is a reserved word in 2GOOD).

The PARENT/child relationship addresses vertical composition, existing between an object (say from class A) and its components, objects in a lower level of abstraction. Note that by definition this is a composition, not an inheritance relationship. In other words, at execution time any A object is the PARENT of all its components. These components are in turn objects, and may have message sending behavior. What makes pronouns such as PARENT useful is the ability we have of designing classes modeling objects that propagate their results by sending messages to their parents, in statements as

```
PARENT < - ResultIs (value);
```

which could be the way a calculator informs the context in which it is placed (its parent, the object that contains it) of its current result. A calculator class designed with this message sending behavior could safely be reused in any context, without the need for changing a single line of code. It is the responsibility of its parent object to handle messages such as ResultIs: a payroll system could use this result to change a salary, and a spreadsheet object could use it to change the contents of a cell.

PARENT/child is a one-to-many relationship, established at parent's creation, and fixed from then on. Each parent may have several children, all objects declared in the object structure section of the class modeling the parent (its components). As usual, parents know the identities of their children, either contained or accessed, and can then request services directly. Each child, on the other hand, has only one parent, does not know its identity, but may use the PARENT pronoun to send messages carrying request results.

We conclude this section with a few remarks on the implementation of the pronoun-based relationship described above. Initially it must be kept in mind that message sending statements to PARENT, as the one shown above, are generated in 2GOOD directly from object communication and scenario diagrams. In 2GOOD, detailed implementation takes place when transforming diagrams to DDL code, and then DDL code to C++ code, where the semantics described above are enforced. The C++ implementation is simple: to hold an access to its PARENT, each object has an extra component in its structure. The value of this component is set when the containing object is created; each of its components now has a parent. Since this relationship is immutable, the cost of its implementation seems negligible, when compared to its benefits.

4. An example

We will use an example based on the home heating system (Hall and Weedon, 1993) to illustrate the modeling of MIL features in 2GOOD, especially PARENT sending messages. We first present the MIL description of a home heating control system and then the 2GOOD implementation of the system. In the diagram shown in Fig. 1, (P) indicates procedural behavior, and (M) indicates message handling. [3] A home heating client initializes the system by setting a temperature and an alarm interval, applying to a home heating object the procedures SetTemperature and SetAlarmInterval. The home heating object in turn tells its configurable timer to set its alarm, and keeps the temperature as part of its state. From this moment on the timer sends to its parent (the home heating object), at the specified interval, the asynchronous message Alarm. Receiving this message, the home heating object sends to its thermometer a request for the current

[3] For the moment please ignore all client messages arriving and leaving the home heating system object. We will return to them later.

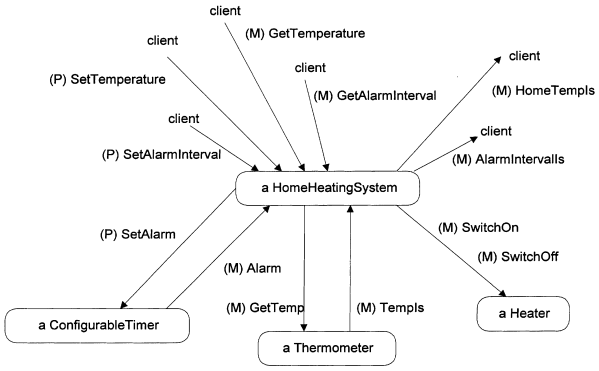

Fig. 1. Object communication in the HomeHeatingSystem.

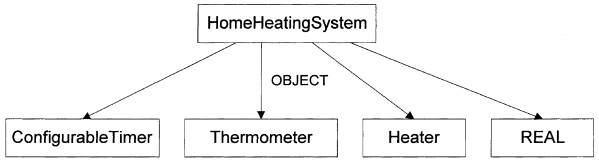

Fig. 2. A static view of HomeHeatingSystem.

temperature, receiving back the message TempIs carrying that temperature. Handling this message, the home heating object activates the heater, sending its state and current temperatures. Responding, the heater is switched on or off.

The MIL instance depicted below describes home heating interconnections in this system: *imports* are resources obtained from other objects, *exports* are resources provided to other objects, and *contains* lists the component objects needed.

- *exports*: GetTemp to Thermometer, SwitchOn to Heater, SwitchOff to Heater, SetAlarmInterval to Timer, GetTemperature to client, GetAlarmInterval to client
- *imports*: SetTemperature from client, SetAlarmInterval from client, Alarm from Timer, TempIs from Thermometer
- *contains*: ConfigurableTimer, Thermometer, Heater, REAL

A 2GOOD implementation for this system follows. We define four classes (Fig. 2), modeling the system itself, the thermometer, the heater, and the timer. The built-in class REAL models the set temperature. We use the same inheritance relations indicated in Hall's paper.

The class HomeHeatingSystem, encoded in a DDL-like notation below, models our system. The OBJECT section models the MIL *contains* clause, listing all component objects. With the exception of REAL and SHORT, built-in classes, classes listed in the application (see below) model all other system objects (actually, as is usually the case of real-time systems, these classes model objects representing hardware devices). Execution starts with the application of a Main procedure (necessary in the main system class) to a HomeHeatingSystem object, automatically created.

```
CLASS HomeHeatingSystem;
  – establishing PARENT/child relationship:
  OBJECT
    ConfigurableTimer        timer;
    Thermometer              thermo;
    Heater                   heater;
    REAL                     temp;
  END OBJECT
  PROCEDURE Main;
  BEGIN
    – create children (components) timer, thermo, heater and temp
  END PROCEDURE
  – to be continued below
END CLASS
```

The code below corresponds to the *imports* section of the HomeHeatingSystem class. To each service listed in the *imports* clause of the MIL description above there corresponds an operation in the class. Some take the form of procedures, since synchronism is needed; others are handlers of asynchronous messages, as the Alarm message sent from the timer component.

```
– services provided to client (imports clause):
PROCEDURE SetTemperature (IN REAL set_temp);
BEGIN
    temp := set_temp;
END PROCEDURE
PROCEDURE SetAlarmInterval (IN SHORT interval);
BEGIN
    timer <- SetAlarm (interval);
END PROCEDURE
– services provided to dependents (imports clause):
ASYNC HANDLER Alarm FROM timer;
BEGIN
    thermo <- GetTemp;
END HANDLER
ASYNC HANDLER TempIs (IN REAL t) FROM thermo;
BEGIN
    IF t < temp THEN heater <- SwitchOn;
    ELSIF t > temp THEN heater <- SwitchOff;
    END IF
END HANDLER
```

The services listed in the *exports* clause become statements in HomeHeatingSystem operations, corresponding to services requested from components. We note once again that so far we are not considering services that could be provided to home heating clients.

The classes ConfigurableTimer and Thermometer, listed below, use the message sending facility we are claiming to improve design for reuse. We offer comments and discuss alternatives after the listings are presented. The class ConfigurableTimer inherits Timer (a device representation class with a MESSAGE section listing Alarm), thus allowing timer to send this message to its parent (in our system, the HomeHeatingSystem object):

```
CLASS ConfigurableTimer; INHERITS Timer; – may send Alarm to parent
  – service provided to client (imports clause):
  PROCEDURE SetAlarm (IN SHORT interval);
  BEGIN
    – define the interval between alarm messages
  END PROCEDURE
END CLASS
```

The Thermometer inherits Sensor (another device representation class), and sends its own message TempIs to its parent, when responding to the message GetTemp:

```
CLASS Thermometer; INHERITS Sensor;
  MESSAGE
    TempIs (REAL);
  END MESSAGE
  ASYNC HANDLER GetTemp;
  BEGIN
    – obtain current temperature (t);
    PARENT <- TempIs (t); – message to generic receiver
  END HANDLER
END CLASS
```

We note that the outgoing communication protocols in classes ConfigurableTimer (inherited from Timer) and Thermometer (statement **PARENT <- TempIs (t);**), used as components in this application, are entirely expressed with

asynchronous messages to PARENT, which in this example is a HomeHeatingSystem object – the context in which the objects timer and thermo are declared.

The same component classes could safely be placed in other applications, where their services are needed, without glue code. They are truly black-box classes; all that applications need to know to reuse them is their communication protocols. This is the central message of this paper: with the correct features in design and programming tools, and using appropriate design disciplines, one can achieve reuse quite effortlessly; in our system, with messages sent to generic receivers, as PARENT.

To complete our example: the Heater inherits Device, is internally represented as a Boolean, and provides SwitchOn and SwitchOff to its clients:

```
CLASS Heater; INHERITS Device;
  OBJECT
    BOOLEAN switched := FALSE;
  END OBJECT
  – services to clients:
  ASYNC HANDLER SwitchOn;
  BEGIN
    IF NOT switched THEN switched := TRUE;
    END IF
  END HANDLER
  ASYNC HANDLER SwitchOff;
  BEGIN
    IF switched THEN switched := FALSE;
    END IF
  END HANDLER
END CLASS
```

5. Alternatives

At this point it is important to compare the ability of sending messages to generic receivers, automatically related to an object, to other ways of allowing the server to communicate its results. The traditional solution, actually the only one available in most OO systems, is to use result parameters in the definition of server procedures, to be sent to the client at termination. This is fine and necessary at times, but as a first drawback it increases parameter lists, a source of programming mistakes, and also increases client–server coupling. More importantly, it imposes a high degree of synchronism: the client is blocked until the operation terminates. As mentioned above, other semantics are possible for operations, fully backed by modern operating systems. Asynchronism should be used when the situation presents itself, relaxing execution constraints that actually may not exist in the domain being modeled.

Another solution is to make the client known to the server by sending the client as an operation argument, when the server is called upon. Again this solution increases parameter lists. The serious problem, however, is that the server, having access to the client, can for example pass it on to other objects, as arguments in their invocations, and this could easily lead to unpredictable results. In 2GOOD a generic client (as PARENT) can only be used, in the definition of server classes, as asynchronous message recipients. There is no way a server can propagate a client, as this would defeat the design of reusable classes.

We argue that the communication protocol we are using has a larger impact on the reuse of classes and in the quality of designs than solutions commonly found:

- it does not prevent such solutions (result parameters in highly synchronized operations) from being used;
- it takes full advantage of resources available in modern platforms, such as threads and message queues, for example increasing the use of parallelism;
- it allows the construction of black-box classes and libraries, reusable in other contexts without code adaptations.

6. Applications as components

We now go back to the object communication diagram shown in Fig. 1 above, and apply our design guidelines to the HomeHeatingSystem class itself, preparing it for reuse. To accomplish this we need to define in what way home heating objects could be useful; what kind of information they must provide to be so. These correspond to the dotted

arrows going to client and arriving at the home heating object, in Fig. 1. We conclude this example by inserting in the HomeHeatingSystem class a message section declaring the messages HomeTempIs and AlarmIntervalIs, which home heating objects can send to their clients, and by declaring more services in this class, namely handlers for the imported messages GetTemperature and GetAlarmInterval. We illustrate this code segment below.

```
MESSAGE
    HomeTempIs ( REAL );
    AlarmIntervalIs ( REAL );
END MESSAGE
ASYNC HANDLER FOR GetTemperature;
BEGIN
    PARENT <- HomeTempIs ( temp );
END HANDLER
ASYNC HANDLER FOR GetAlarmInterval;
BEGIN
    – get alarm interval (either keep as component, or request from timer,
    – adding operations to its class)
    PARENT <- AlarmIntervalIs ( interval );
END HANDLER
```

The HomeHeatingSystem class is now a black-box, and can be reused in any context without code adaptations. It does provide services to possible clients, but always through messages to PARENT. Any client can have a home heating object as component, but should mind the MESSAGE declaration section shown above, as this defines home heating communication protocols.

7. Discussion and conclusions

As well pointed out by Biggerstaff, MIL proposals do not really mitigate the library scaling problem, although proposals like OOMIL, which combines OO and MIL, may help in attacking this problem. Using a MIL approach during the design of classes increases the visibility of the important aspect of composition, in such a way that information hiding and separation of concerns may be better deployed by designers. A MIL description should clarify the composition being used.

The accompanying design method/tool should stress that well-defined interfaces are a result of a good abstraction/generalization/parameterization process, which in turn depends on domain analysis. Although this is a fact, a domain analysis is no good if the designer does not compose the system in an organized manner, with well-defined (and perhaps parameterized) components, and interchangeable interfaces. As such, designing for reuse must have proper representations. Design patterns and realms (Batory et al., 1994) are good examples of how to use generalization and abstraction during the design of object oriented systems. Ideally, perhaps, a good design should be expressed in MIL's using design patterns as building blocks. Design patterns linked by MIL descriptions provide a well-suited infrastructure for reuse, since the encapsulation of design decisions will be cast in a generalization framework and will be described both in terms of composition and interfaces.

Designing for reuse may also be influenced by the programming language to be used. If this language does not have capabilities to express MIL based designs, our effort to make reusable components will be counter-balanced by the effort we will spend to implement reusable components. One of the advantages of 2GOOD and DDL is that they have been designed to describe the same OO discourse, and provide automatic transformations from 2GOOD diagrams to DDL code. This supports formalization and traceability, allowing system construction with no discontinuities.

The fact that a 2GOOD system, a collection of classes, has no global referencing environment or external functions can be seen as an additional facility for making possible the implementation of blind servers. Providing the possibility that a given collection of classes be completely independent with respect to whom and where it will be used is truly the essence of black-box reuse. The generic PARENT capability of 2GOOD was designed exactly with this purpose, to simplify the black-box reuse of classes or class collections, for example mediated subsystems.

We understand that black-box reuse will be successful depending also on the quality of the domain analysis process as well as how the organization of the related design knowledge is put together, but the possibility of implementing blind servers, and thus avoiding glue code, have appealed to us as a important contribution in the design for reuse

discipline. Alternatives to the PARENT mechanism, discussed above, do not seem to provide the same component independence we are offering. This independence naturally suggests an improved bottom-up design process: each class imports messages sent by their components, via asynchronous handlers, and defines the messages it in turn exports to its parents.

It is also important to stress that the decoupling facilities presented above can (and should) be used, in modeling an application, at all development levels. For example, in drawing an object communication diagram, special PARENT objects can be used. During the automatic code generation we provide, the classes modeling objects sending messages to those special designators are contemplated with MESSAGE sections where those messages are listed. Likewise, PARENT objects can be used in scenario diagrams. These are concrete contributions to design for reuse.

With regards to future work, we plan to attack the aspect of evolution (Goguen, 1996), which was not addressed in this paper; since we are focusing on black-box reuse, the management of modifications on MIL descriptions was not a central issue. Other fronts are the study of other forms of generic relationships, such as CREATOR/creature and SENDER/receiver, the consideration of other message semantics (futures and handshakes) as decoupling mechanisms, and the discovery and cataloguing of design patterns using generic receivers. We are also considering the impact that the message passing features of 2GOOD may have on modern software generators, for example GenVoca and Draco-Puc (Leite et al., 1994).

References

Batory, D., Singhal, V., Thomas, J., Dasari, S., Geraci, B., Sirkin, M., 1994. The GenVoca model of software system generators. IEEE Software, 89–94.

Biggerstaff, T.J., 1994. The library scaling problem. In: Proceedings of the Third International Conference on Software Reuse. IEEE Computer Society Press, Silver Spring, pp. 102–109.

CACM, 1997. Communications of the ACM, (special issue).

Carvalho, S.E., Cruz, S.O., Oliveira, T.C., 1996a. 2GOOD: A Tool for Second Generation Object-Oriented Development, Project ARTS, Subproject 2GOOD, Technical Report No. 1.

Carvalho, S.E., Cruz, S.O., Oliveira, T.C., 1996b. The Design Description Language, Project ARTS, Subproject 2GOOD, Technical Report No. 2.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. Design Patterns: Elements of Object-Oriented Software. Addison-Wesley Professional Computing Series, Reading, Massachusetts.

Goguen, J.A., 1986. Reusing and interconnecting software components. IEEE Software, 16–28.

Goguen, J.A., 1996. Parameterized programming and software architecture, Keynote address. In: Proceedings of the Fourth International Conference on Software Reuse. IEEE Computer Society Press, Silver Spring, pp. 2–10.

Griss, M.L., 1994. Panel: Architecting kits for reuse. In: Proceedings of the Third International Conference on Software Reuse. IEEE Computer Society Press, Silver Spring, pp. 216–217.

Hall, P., Weedon, R.,1993. Object oriented module interconnection languages. In: Prieto-Diaz, Frakes, (Eds.), Advances in Software Reuse. IEEE Computer Society Press, Silver Spring, pp. 29–38.

Leite, J.C.S.P., Sant'Anna, M., Gouveia, F., 1994. Draco-Puc: A technology assembly for domain oriented software development. In: Proceedings of the Third International Conference on Software Reuse. IEEE Computer Society Press, Silver Spring, pp. 94–100.

Neighbors, J., 1994. An assessment of reuse technology after ten years. In: Proceedings of the Third International Conference on Software Reuse. IEEE Computer Society Press, Silver Spring, pp. 6–15.

Nierstrasz, O., Tsichritzis, D. (Eds.), 1995. Object-Oriented Software Composition. Prentice-Hall, London.

Prieto-Diaz, R., Neighbors, J., 1986. Module interconnection languages. J. Systems and Software 6 (4), 307–334.

Shaw, M., Garlan, D., 1996. Software Architecture. Prentice-Hall, London.

Tracz, W., 1993. A parameterized programming language. In: Prieto-Diaz, Frakes, (Eds.), Advances in Software Reuse. IEEE Computer Society Press, Silver Spring, pp. 66–79.

Sergio Carvalho got an Electrical Engineering degree in 1961, from the Catholic University in Rio. After working in industry for six years, he came back for a M.Sc. degree in Computer Science also at PUC-Rio. Immediately upon completion, he went to the University of Waterloo for his Ph.D., which he completed in 1974. Since then he has been working at PUC-Rio, as an associate professor. He received an IBM World Trade scholarship in 1976, and spent a year at the San Jose research laboratory. He then got an associate professor position at the University of Colorado in Colorado Springs, where he stayed for periods of two and one year, returning then to PUC-Rio. His main interests are in programming languages, object and agent orientation and software development methods.

Julio Cesar Sampaio do Prado Leite (http://www.inf.puc-rio/br/~julio) is an Associate Professor at the *Departmento de Informática* (Informatics department) of the Catholic University of Rio de Janeiro (PUC-Rio) and director of the Draco-PUC project. Dr. Leite is an active researcher in the areas of: software reuse, requirements engineering and reverse engineering. He is member of the IFIP 2.9 working group (Software Requirements Engineering) and of the IEEE sub-committee on software reuse. He has published more than 40 full articles in conference proceedings and 5 journal papers. Dr. Leite got his Ph.D. from University of California, Irvine in 1988 and is member of the ACM, the IEEE Computer Society and a founding member of the SBC *(Sociedade Brasileira de Computação)*.