A Visual Object-Oriented Meta-CASE Environment

Alvaro Ortigosa* - Marcelo Campo* - Roberto Tom Price*

Universidade Federal do Rio Grande do Sul - Instituto de Informática
Caixa Postal: 15064, Porto Alegre, Brasil
***Universidad Nacional del Centro de la Provincia de Buenos Aires**
Facultad de Ciencias Exactas - Instituto de Sistemas Tandil
San Martin 57, (7000) Tandil, Argentina
email: {ortigosa,mcampo,tomprice}@inf.ufrgs.br

Abstract

The current need of highly integrated and customizable software development environments requires a "meta" approach. An object-oriented approach that supports visual specification of languages and relationships between tools is presented. This Meta-CASE approach, based on an object-oriented editor framework, includes all the features encountered in earlier language oriented editor generators and in the current database and hypertext approaches to environment integration.

1. Introduction

A CASE environment is composed by tools to create and transform documents of different types. An important class of tools are graphical editors that support the creation and manipulation of different document types, defined by the method implemented by the environment. The editors can be specially build for each method notation or can be generated by a meta-editor, which receives the specification of the language syntax and semantics and generates a specific editor for that language. A meta-editor is an important component of the so called Meta-CASE environments.

These environments support the specification of the development method through the use of some formalism to describe the structure of the different types of documents and their relationships. A document type will be manipulated by a specific editor that has knowledge of the syntactic structure and static semantics of that document type (e.g., DFD). The environment often supports consistency among the different documents produced in the development process through a common data dictionary and tools that know the method semantics [Brown 92].

---

* This work is partially supported by FAPERGS and CNPq through the AMADEUS project at the Instituto de Informática - UFRGS, Porto Alegre, Brasil.

This work presents an approach to Meta-CASE development that combines a high level visual notation to describe the document structures with visual composition of object-oriented frameworks. The approach is based on the idea that the application of a development method produces a set of interrelated documents which can be implemented as a hyperdocument [Cybulsky 92]. The specification of the generic structure of this hyperdocument with an object-oriented semantic data model allows to visually model the development method as the authoring process of the hyperdocument.

The model unifies in a simple graphical formalism the specification of the document internal structure, as a set of interrelated objects, as well as inter-document relationships. A specific environment is viewed as an ***object-oriented hyperdocument management system***, that has knowledge of the semantics of the development method. With this view, a Meta-CASE object-oriented framework provides the generic abstractions needed to produce a specific development environment. The approach supports the specification of a method as a visual object composition process.

This article is organized as follows. First the conceptual framework on which the work is based is defined. Next a prototype of a tool implementing the approach is described and a brief example of an environment generation for the OOSE method [Jacobson 92] is shown.

2. The Conceptual Framework for Meta-CASE Development

System development using any method produces a set of interrelated documents, each one describing different aspects of the reality being modeled. The domain application entities are described in their structural, behavioural or functional aspects, using different techniques or languages specially designed to highlight some of these aspects (e.g., ERD, Statecharts, DFD). Thus the *same entity* is represented in different documents using specific syntactic objects defined by the notations (e.g., files and entities, or attributes and states). For instance, in the OOSE method, the same entities that are defined as Objects in the Requirements and Analysis Models, are viewed as Modules in the Design Model.

Therefore, if a document describes partially a set of entities, and each entity may be described in different documents in diverse ways, each document may be considered as showing views of the application domain entities.

In a development environment, the information about these entities will be stored in some kind of persistent data repository. Independently of the storage implementation (centralised common repository, distributed database, etc), the existence of an underlying conceptual environment database is assumed. So, if a document shows views of the domain entities, these documents correspond to views of this database. That is, the database will contain *one* object that represents some domain entity, and each document will show only the attributes and relationships with other entities of the domain, that are relevant to the aspects that this document reflects.

Each document type will have associated manipulation tools. These tools will be used to create and manipulate the documents, according to the structure of that document type. The tools are mainly editors, specially during the first development phases (i.e., analysis, design), but transformational and analytical tools will also handle the documents. In the context defined above, the process of document editing can be seen as a process of updating the database, so, the editors will be used as an **interface for direct manipulation of the**

database. The database determines the correct structure of a document type and the editor must provide a user-convenient way to manipulate this structure.

Under these considerations, in order to specify an editor for a document type it is enough to describe the object model of the documents and the visual appearance of each component.

This approach, the use the object model as a partial definition of the editors, allows to specify with the same notation the generic structure of the hyperdocument produced by the application of a development method. In this way, by describing the hyperdocument structure it is possible to define the method, as is next explained.

The specification of the structure of each document is made using an object-oriented data model based in OMT [Rumbaugh 91]. The document is described as an aggregation of objects, which are the syntactic components of the notation defined by the document type.

Each syntactic component is described by an object class (e.g., process, state, class, module, flow), encapsulating its attributes (names and types) and its behaviour. In order to define the structure of a class of documents, it is necessary to specify what object classes will compose the documents. The definition of the abstract structure is completed with aggregation and grouping relationships. The associations that may or must exist between the objects must be specified too.

Fig. 1 shows an example of a partial document structure definition. The document type defined represents the *Domain Model* used in the OOSE method. The document is defined as an aggregation of instances of the *DomainObject*, *Knows* and *Inherits* classes. The class *DomainObject* has defined an attribute *Name* as being a string.

Relationship-entities provide the support to specify syntactic components whose existence depends on the existence of other components. For example, relationships between classes in a OMT diagram or flows in a DFD, can be represented by this type of entity. In the document type defined in fig.1, the *Knows* and *Inherits* classes represent relationship-entities that will relate instances of the *DomainObject* class. Instances of such classes will exist only when exist the domain objects that they have to relate.

There are several types of predefined associations that allow to visually specify additional information for the meta-editor. An example is the association, called *view-of*, between objects and its (possibly multiple) visual representations. Every object and relationship class may have one or several related graphical objects. These graphical objects encapsulate the context dependent information used to show the components to the user. In fig.1 exists one relationship of this kind between a *DomainObject* and a *DomObjView*. This relationship specifies that each instance of *DomainObject* may be associated with zero or more visual presentations, while a visual presentation will correspond to one domain object.

Associations can be defined between entities of the same document or between entities defined in different documents, representing the hyperdocument structure. In the example of the fig.1, a DomainObject may be related with a UseCase, a component of a UseCases document type, by an *involvedIn* relationship. As this kind of relationship will generate hyperlinks, the method determines the structure of the resultant hyperdocument.

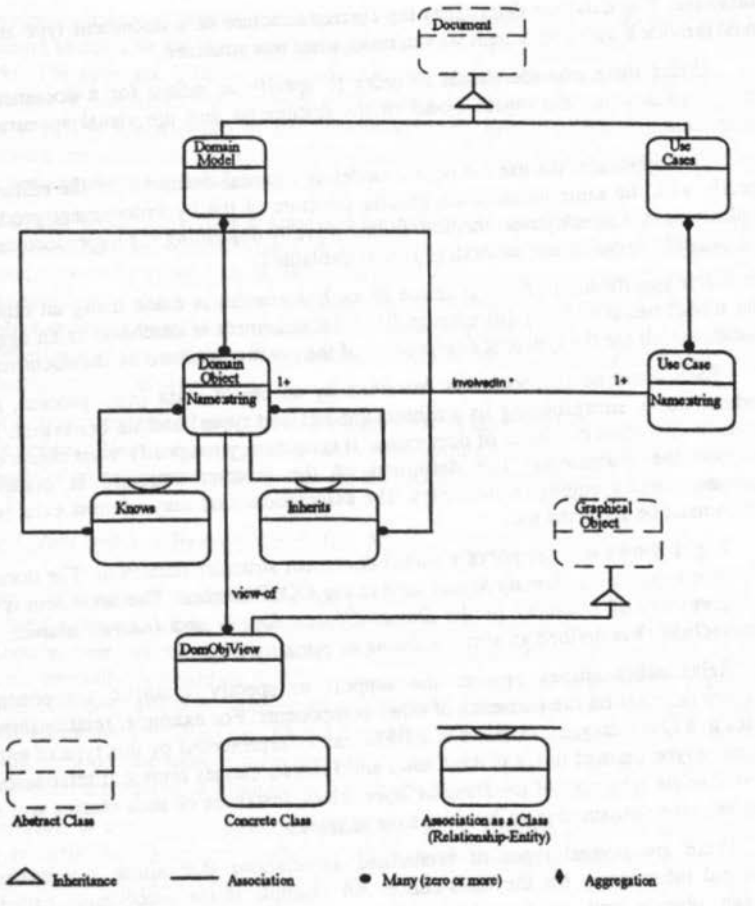

Fig 1. Partial definition of a Domain Model document structure

The functionality of the relationships provides the knowledge about the restrictions on the existence and conectivity of related components. A 0:1 functionality expresses that the relationship is optional. A 1:+ functionality specifies that a determined component will be related to at least one object of the other type. In the example of the fig.1, a DomainObject must have at least one UseCase associated. This constrain reflects that in the method being specified it has no sense to define a domain object that is not involved in any use case. In addition to the relationship functionality, information about the creation order of the related objects is provided. In the fig.1, the * in the *involvedIn* association defines that the creation order of the domain object and the use case is meaningless. However, in other cases this order is important. For example, when a new process specification is given, the associated process must already exist. Fig.2 shows the partial specification of these objects.

This specification describes that a *Process* may be associated with (*described by*) a *Process Specification* (0:1 functionality), but each *Process Specification* must have a corresponding *Process*(1:1 functionality), and this *Process* must be created first ("+" symbol).

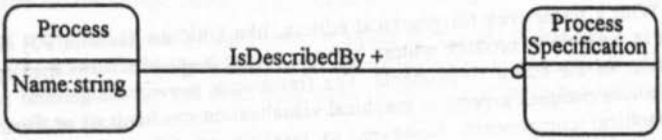

Fig.2. Partial specification of two related entities

Using the information provided by the relationship functionality the meta-environment can infer the order in which different document types must be created to maintain the database always in a consistent state.

2.1. Automatic Consistency Management

The existence of relationships between the documents and, mainly, the presentation of the same entity in different documents, makes necessary to maintain consistency between these documents. There are two consistency types that must be considered: one is the consistency between objects that represent the same domain entity. A change in the object state in one document must be reported to the other documents (or document editors) so that they can reflect the new state. The other consistency type is derived from the relationships between different objects that the development method uses.

Considering the documents as views of the conceptual environment database, some cases of potential inconsistency can be automatically resolved by the environment. When the user changes the contents of a document, these changes are made directly in the environment database, and all the documents will automatically reflect the new state of this database.

For example, if some object is deleted from a model, the corresponding objects in other documents will have to disappear too, and the environment can manage this situation without any especiallised tool.

This functionality implies that the database must have information about how a domain entity is presented in different documents, that is, must know the correspondence between components defined in different description techniques. A special type of association, called *semantic equivalence association*, is used to specify that several component types represent the same entity. A classical example is the equivalence between files of a DFD and entities in the corresponding ER model.

A complete description model must provide all the component correspondences using this type of associations.

2.2. Environment Generation as Visual Composition of Object-Oriented Frameworks

An object-oriented framework [Deutsch 89] is a skeleton of the implementation of a system or application, in a specific problem domain. A framework represents an architecture

for reuse, where the components are represented by abstract classes. The applications are produced by specialising these classes to provide the specific behaviour to the methods that they define but don't implement. The framework is a generic design, that divides the problem in a set of component classes and defining their general behaviour and their communication protocols.

With a framework for graphical editors, like Unidraw [Linton 90] or HotDraw [Beck 91], it is possible to produce editors by specializing a set of classes that defines the generic behaviour of the editor components. The framework provides a generic editor architecture that resolves complex aspects as graphical visualization mechanisms or direct manipulation of the graphical components. However, to produce an editor that supports a determined language, the user must program the classes that implement the structures that represent the language and the rules that determines a valid document.

With the approach described in the previous section, graphical structure editors can be easily generated. The use of an object-oriented model allows to combine the advantages of a high level specification of the document structure with the mechanisms of the object-oriented frameworks to support design reuse [Johnson 91]. An editor framework provides the generic abstractions used by any editor (i.e., document, graphical components, commands, user interface) and the specification process is reduced to visually describe how subclasses of the generic classes defined by the framework are structured.

An editor generator can use the knowledge provided by the relationships defined in the model and the knowledge of the framework structure to generate classes that implement each component type. Using the information provided by the structural relationships (aggregation and grouping) the generator can generate classes that implement the structure defined and perform the semantic checks through a message protocol defined by the framework. Any component class defined in the model will agree with the protocol defined by a framework class, say *Entity*, inheriting its behaviour. This class implements all the methods that are needed to edit and manipulate the structure of objects. The private behaviour of each component, can be specified using a special textual language or the base language of the framework.

The user interface functionality of the editors, can be specified using a similar notation that is used for structure specification. In any editor framework the user interface components are encapsulated by classes, ones implementing the visual presentation mechanisms and others implementing input event handling. The classes that implement visual presentation are connected to the objects that they are presenting in the screen to show the presentation corresponding to the internal state of the object. To accomplish this task, both classes must provide standard methods to state change notifications and to pass values for visualization. The presentations will be directly manipulated by the user, so, the classes implementing the manipulations (e.g., mouse select, dragging, etc.) must agree in a standard protocol with the presentations to implement the event handling on any presentation class. The same considerations holds for the user interface classes and classes that implements specific commands of the editor, for example object deleting or link traversal.

If two or several classes agree in a known protocol between them, it is possible to describe visually how instances of each class are interconnected to compose the application, using a graph-based notation. Therefore, using the idea of visual object composition the document structure and the editor architecture can be described using a similar notation.

Viewing the specification process as visual object composition, it is possible to generalize the environment specification to include other aspects involved in a software development environment. A Meta-CASE framework that defines the standard protocol through the different components collaborates to implement the functions of the environment. Once an enough number of components has been developed, the visual composition strategy can be used to specify the composition of a particular environment.

The visual composition can be supported by an editor for the linking of object classes and instances. This editor can be generated as an specialised version of the editor framework implementing a visual language for describing object composition, therefore implementing a Visual Scripting Meta-Tool.

3. An Experimental Prototype

At the current stage of the project, a prototype to test the ideas presented above has been developed. Our main objective was to design an environment that allows the user to visually specify the hyperdocument structure, the visual presentation of each document component type, and the user-interface of the document editors. These three aspects are supported by graphical editors based on an object-oriented framework for graphical editors.

This framework supports the common functionality of the generated editors, as structural editing model, based on *placeholder* substitutions to edit object attributes, and hypertext functionality as described in [Ortigosa 94].

In order to generate a new editor, three steps must be carried out. First, the document structure must be defined. This is made using an editor that allows the definition of the structure using the formalism described in §2, with some additional components to make easier the user task. Special icons were added to represent objects of special types, like documents or graphical objects. The use of one of these special icons implicitly specifies that the object inherits from the corresponding abstract class.

Fig.3 shows a snapshot of the definition of the example already presented. There, some new symbols are used: the symbol with name *DomainModel* represents the document being defined The symbol with name *UseCase* represents an object that belongs to another document type, an *UseCases* document, in this example. From the *"involvedIn"* association is derived a hypertext link between the objects of *DomainObject* and *UseCase* types. Each link type is implemented by a class (*involvedIn*, in this case), and specific functionality can be obtained subclassing a generic link class.

The editor generated from this description will allow to edit objects of the *DomainObject* type, and link them by relationships (graphical links) of type *Knows* or *Inherited*.

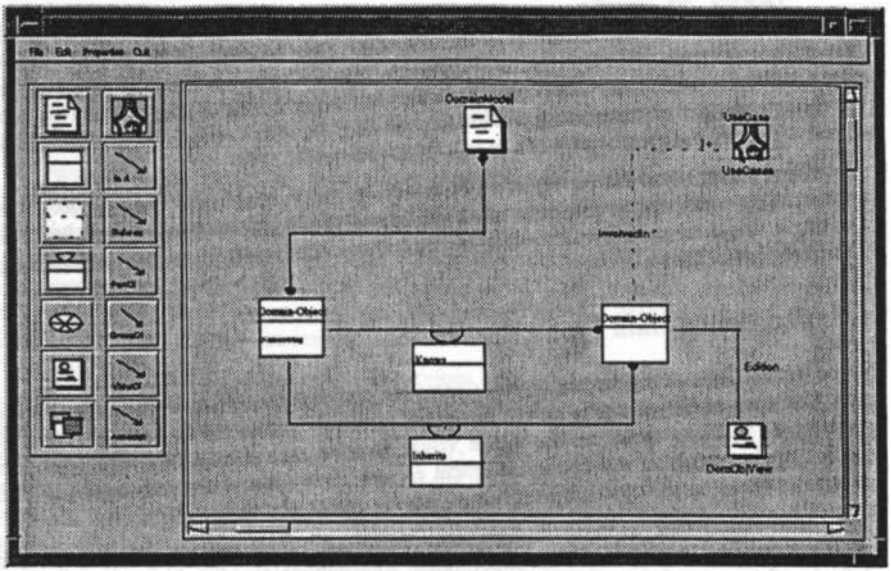

Fig 3: Definition of a Domain Model editor/document for OOSE

Fig. 4 shows an example of the process of link creation. Editing a document of type *UseCases*, the user selected the *Uses* relation from the menu, and then selected the *Operator* user as origin of the link. At this point, the editor indicates the valid ends of the link, according with the document type definition. Here, there are two objects that can be linked to *Operator*, *Return-Item* and *Change-Item*.

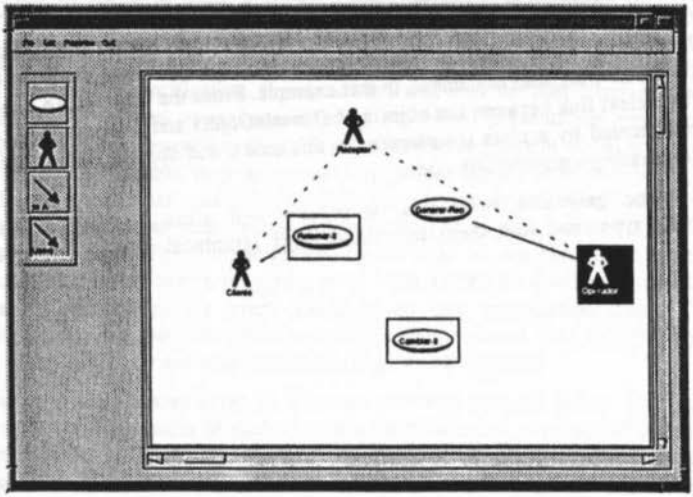

Fig. 4. Snapshot of an editor in link creation operation.

In any editor, the user will be able to create generic hypertext links (called *user links*) between any two edited objects, and links between objects and annotations (*annotation links*). In the editor generated from the definition of fig. 3, in addition he/she will be able to create links of type *involved*, from a *DomainObject* to an object of type *UseCase*, component of a *UseCases* document.

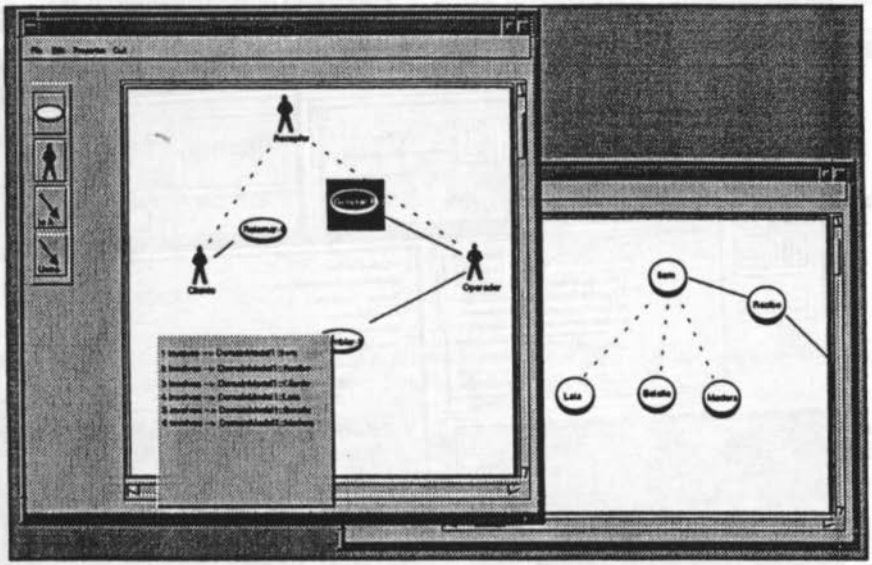

Fig 5: Generated editors for UseCase and Domain Model document types, related by hypertext links

Fig. 5 shows an example of hyperlink creation. In a UseCases document, the user selected the use case *"Generate-Report"* as the origin of a hyperlink. Among the possible alternatives (annotations, *involves*, user link), the user has chosen the *involves* type. This link type was defined to have as origin an object of type *UseCase*, and as a target, an object of type *DomainObject*. So, the environment shows to the user the existent objects that can be target of an *involves* link, and asks him/her for a selection.

In this way, the developer is assisted and guided through the relationship creation. Furthermore, the environment can guarantee that the associations created by the user are consistent with the underlying development method.

The second step needed in order to generate an editor is to describe each graphical object associated with the document components. This description is made using a special editor that allows the user to draw the wished object appearance. This editor is a specialisation of the editor framework, with classes that support the interactive drawing of the graphical appearance of each syntactic component of the document type being defined.

The third step is the specification of the editor user interface. This specification covers two aspects: the user interface composition and the manipulation functionality that will be provided by the editor. The specification of both aspects is made using an editor that supports the visual composition of interface objects [Campo 94]. The visual language was designed to

describe how the user interface of an editor being generated is composed by instances of classes defined in the library of components of the framework.

Fig. 6 shows a snapshot of the editor and associated editors showing code and documentation of the framework. This version was generated using the meta-editor to specify a hyperdocument that binds the user interface components of the framework to the code that implements these components.

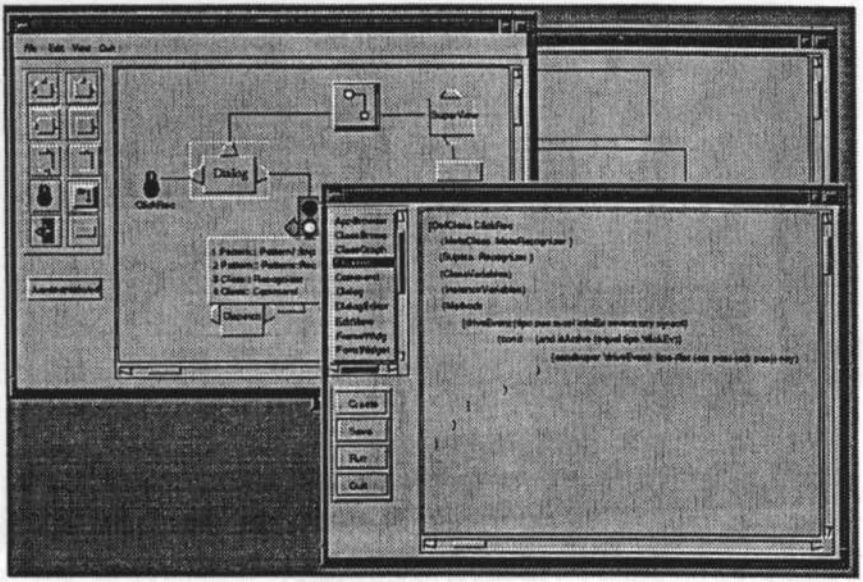

Fig 6: Visual Composition of the User Interface

4. Conclusions and further research

A software development environment specification model that combines the advantages of visual specification with the specialisation facilities provided by an object-oriented framework for the creation of editors was presented.

Different views of the software objects being created and handled by the environment are shown as different documents. As the objects are interconnected by several types of relationships, the collection of documents of the environment forms a hyperdocument. The user interacts with the documents (i.e., the views of the software objects) with editors that really are direct manipulation interfaces to the software object database.

The experimental prototype has shown the easiness of building of a new environment, as well as the simplicity of the addition of new functionality through the specialisation of framework classes.

The formalism is powerful and flexible, resulting in simple specifications that are easy to understand, but it has some limitations, such as lack of tool, user and activity concepts. Further research must tackle the problems of fully incorporating these concepts (now dealt

with only implicitly) into the notation. An explicit knowledge of the activity concept is necessary in order to provide a complete control over the development process. In addition, validation procedures, management reports and other method independent controls must be allowed to be defined.

The prototype also must be extended in several development environment dimensions, which we are now looking at, like automatic transformation mechanisms, metrics collection, quality auditing, improved visual mechanisms, support for groupware, object distribution and improved software re-use mechanisms.

5. Acknowledgments

The authors would like to thank Esteban Pastor for his help in the development of the graphical portion of the environment prototype.

6. References

[Beck 91] K. Beck, HotDraw: **A Framework for Semantic Graphic Editors,** Position Paper for the Architecture Handbook Workshop, OOPSLA'91.

[Brown 92] A. Brown, Feiler P, Wallnau K., **Understanding Integration in a Software Development Environment,** CMU/SEI Tech. Rep. 91-TR-31. also appeared in Procs. of the 2nd IEEE Intnl. Conf. on Systems Integration, pp 22-31, Morristown, NJ, June 1992.

[Campo 94] M. Campo, **Visual Instantiation of Object-Oriented Frameworks,** TI N° 387, UFRGS, Instituto de Informática, CPGCC, February 1994. (in Portuguese).

[Chen 92] M. Chen-R. Norman, **A Framework for Integrated CASE,** IEEE Software, March 1992.

[Cybulsky 92] R. Cybulsky, **A hipertext-based metacase environment,** IEEE Software, March 1992.

[Deutsch 89] P. Deutsch, **Frameworks and Reuse in the Smalltalk-80 System,** in Software Reusability, Vol. II, C. Pearlis and T. Biggerstaf Eds., ACM Press, 1989.

[Jacobson 92] I. Jacobson, M. Christeron and G. Overgaard, **Object-Oriented Software Engineering: A Use Case Driven Approach,** Addison-Wesley, ACM Press, 1992.

[Johnson 91] R. Johnson - V. Russo, **Reusing Object-Oriented Designs,** Univ. Illinois at Urbana Champaign, Tech. Rep. UIUCDCS91-1696, 1991.

[Linton 90] R Linton. - J.Vlissides, **Unidraw: A Framework for Building Domain-Specific Graphical Editors,** ACM Trans. on Information Systems, Vol 8, N.3, July 1990.

[Ortigosa 94] A. Ortigosa, **Incorporating Hypertext in the Software Development Environments,** TI N° 396, UFRGS, Instituto de Informática, CPGCC, February 1994. (in Portuguese)

[Penedo 93] L. Penedo, **Process Based Software Engineering Environments,** Tutorial Notes, Brazilian Symposium on Software Engineering, Rio de Janeiro, October 1993.

[Rumbaugh 91] J. Rumbaugh, et. al, **Object-Oriented Modelling and Design**, Prentice Hall, 1991.