Automatic ObjectPascal Code Generation from Catalysis Specifications

João Luís C de Moraes **Antonio F do Prado**
moraes.uol@uol.com.br prado@dc.ufscar.br
Universidade Federal de São Carlos – UFSCar,
Departamento de Computação – DC,
São Carlos – SP, Brasil, CEP 13565-905

ABSTRACT

This paper presents a Component-based Framework Development Process, of the Cardiology Domain. The Framework, called FrameCardio, was developed in 4 steps: 1-*Problem Domain Definition*, *2-Components Specifications*, *3-Components Internal Design and 4-Components Implementation*. In the first step the framework requirements were identified, based on experiences in the development of a cardiology system with 320 classes. The main models specified in this step are the Use Cases, Actions and Collaborations Models. In the *Component Specification* step component external behaviors were defined with their responsibilities, operations and interfaces. The main models are the Types Model, that describes an object external behavior, regardless of implementation decisions, and the Component Interactions that details the behavior of each case used in the Sequence Diagrams. Right after, in the third step, the Specified Components are refined, considering the implementation technologies. Among the models of this step, the components Classes Diagram, Components and the Component Packages stand out. The modeling was supported by a CASE tool. Finally, in the *Components Implementation* step the components code was generated in the *ObjectPascal* language, using a Transformation System.

FrameCardio was structured in layers and organized in components packages, available in the CASE tool, to be reused by the applications. In the same way as in the development of FrameCardio, the Transformation System is used to generate the *ObjectPascal* code of the applications. A Cardiology domain Application is presented to show Framework components reuse.

Key words: Transformation System, Framework, Catalysis, Component-Based Development.

1 *INTRODUCTION*

The reuse is an essential principle in Software Engineering to assure the reduction of efforts and costs in the Software Development and code redundancy. Different Software Development Processes have been researched to improve the software production. Researches have been exploring different technologies, including the use of CASE tools (Computer-Aided Software Engineering), frameworks, Software Transformation System and object-oriented programming languages, to obtain high quality software with affordable cost.

Aiming to improve the Software Development Processes, this paper presents a Component-Based Framework Development, accomplished in four steps - development *for* reuse and development *with* reuse. The first three steps, responsible for modeling the framework, are accomplished supported by a CASE tool, and correspond to the three levels of the Component-Based Development Catalysis Method [1]. The fourth step is responsible for the Components Implementation in *ObjectPascal* language [2], and it is supported by a Software Transformation System, called Draco-PUC [3, 4]. The framework components can be reused (development *for* reuse) in the Cardiology domain applications, facilitating the modeling, reducing the code redundancy and the costs of the maintenance.

To facilitate the applications development (development *with* reuse), reusing the framework components, the CASE tool Rational Rose 2001 [5] is also used, for modeling, and the Draco-PUC Transformation System to generate ObjectPascal code. The code in the ObjectPascal language is generated from the descriptions of the applications project specifications. The class structure and the component interface codes are generated with their atributes and methods prototypes. Based on the code, specified by the Software Engineer, for the methods behavior, we can have a more complete implementation of the applications.

To support the development of the framework FrameCardio and the Applications, different technologies are used:

a) The Component-Based Software Development Catalysis Method;

b) Draco-PUC Transformation System, for ObjectPascal code generation; and

c) The Object-oriented language, ObjectPascal, for implementing the framework FrameCardio componentes applications, according to the Software Development Processes proposed.

This paper is organized in the following way: Section 2 presents the main technologies integrated in the development process; Section 3 presents the Component-Based framework Development Process, for a Cardiology domain; Section 4 presents a Case Study, of an application reusing FrameCardio; and, finally, Section 5 presents a conclusion of this research project.

2 MAIN TECHNOLOGIES OF COMPONENT-BASED SOFTWARE DEVELOPMENT PROCESSES

The Catalysis Method is presented, used in the Framework and Applications modeling, in the CASE tool Rational Rose 2001.

2.1 Catalysis Method

Catalysis is a Component-Based Software Development that integrates techniques, Patterns and Frameworks. Catalysis began in 1991 with of OMT and had influences of the methods Fusion [6] and UML [7, 8]. It supports the characteristics of the oriented-object technologies like Java, CORBA and DCOM and its notation is based on Unified Modeling Language (UML).

Catalysis is founded on three principles: **Abstraction, Precision and Pluggable parts.** The **Abstraction** guides the Software Engineer in search of the essential aspects of the system, deferring details that are not relevant for the system context. The **Precision** aims to reveal mistakes and inconsistencies in the modeling and the **Pluggable parts** seek the reuse of components to build other components [9].

Software Development Processes in *Catalysis* follow the characteristics of the Spiral model of the Software Engineering [10], and is divided in three logical levels: ***Problem Domain, Components Specification and Components Internal Design***, corresponding to the traditional activities of the software lifecycle: **Planning, Specification, Design and Implementation**, that are executed in an incremental and evolutionary way, resulting in the generation of a new version of a prototype to each accomplished cycle.

To facilitate the modeling, according to Catalysis, the CASE tool used is Rational Rose 2001.

2.2 Draco-PUC Transformation System

In the Draco-PUC Transformation System, a domain is composed of three parts, ***Language, Prettyprinter and Transformers***, according to Figure 1. The language is defined through its grammar and its parser, that analyze any program of the domain, and generates its internal representation in Draco-PUC. This internal representation, named Draco Abstract Sintaxe Tree (DAST), is used to apply the transformation components to generate a new DAST in the same or in another domain. The prettyprinter is responsible for showing the

DAST, in the textual form, oriented by the domain language syntax. The transformers are transformation components packages that act in a DAST, to generate a new DAST, according to Figure 1.

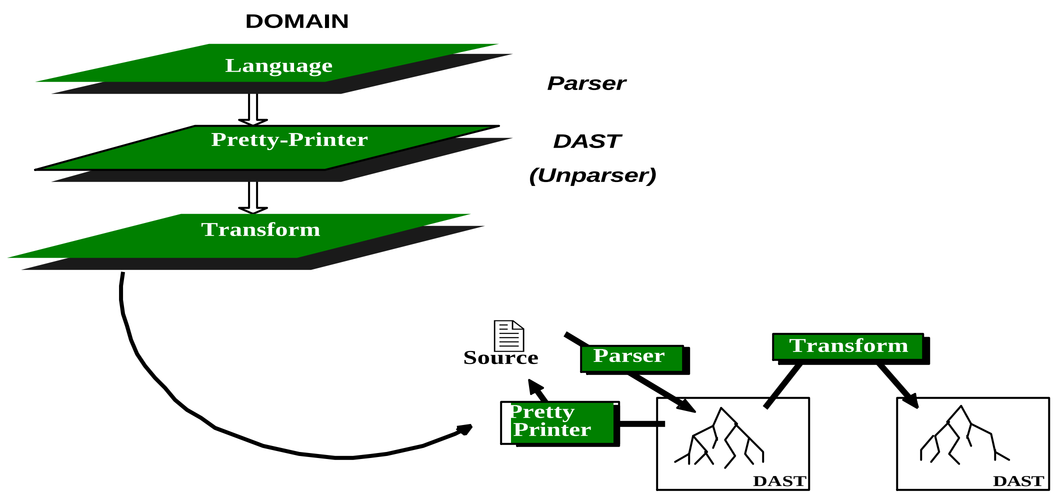


Figure 1: Parts of a domain in Draco-PUC Transformation System.

The Draco-PUC Transformation System was used as a main mechanism for the code generation of FrameCardio and their applications. To support the code generation two domains have been built: *MDL* (Modeling Domain Language) and *ObjectPascal*. The UML specifications, modeled in the CASE tool, according to Catalysis, are stored in a textual descriptions file. Based on these textual descriptions language, the *MDL* modeling domain has been built. To generate the ObjectPascal code from *MDL* descriptions, the *ObjectPascal* domain has also been built. The transformations that generate code, are based on the grammars of these both domains, *MDL* and *ObjectPascal*, and are partially presented in Figure 2.

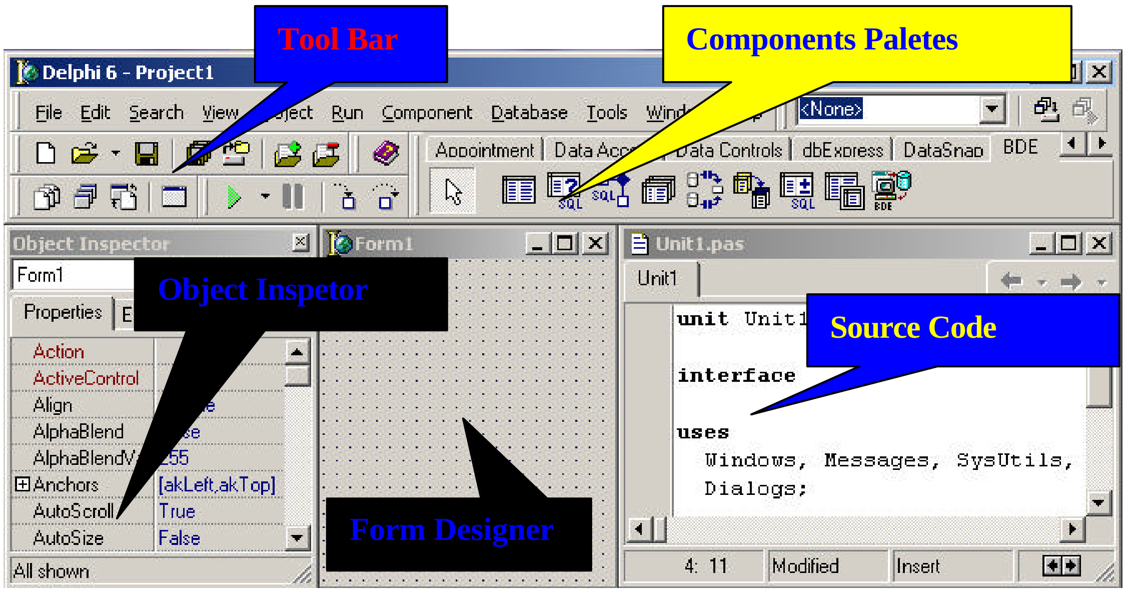
MDL	*ObjectPascal*
class_Object : *'CLASS'* STRI classAttributes; **classAttributes :** quidu .nl stereotype | quid classAttributes_Attr ; **classAttributes_Attr :** (documentation)? (stereotype)? (superClasses)? (used_nodes)? **superClasses :** *'superclasses'* .nl inheritance_relationship_list*; **inheritance_relationship_list**: *'(list'* .sp *'inheritance_relationship_list'* inheritance_Relationship* *')'* ; **inheritance_Relationship :** *'(object'* .sp *Inheritance_Relationship'* .nl attributes? .nl quid .nl stereotype? .nl label? .nl supplier .nl quidu? *')'*; **used_nodes:** *'used_nodes'* .sp uses_relationship_list;	**compilation_file :** program_file | unit_file | library_file | object_form | package_file ; **package_file :** *'PACKAGE'* .sp IDENTIFIER *';'* requires_clause contains_clause ; **requires_clause :** */* empty */* | *'REQUIRES'* .sp requires_units_list *';'* ; **unit_file :** unit_heading interface_part? implementation_part? initialization_part? *'.* *';* **unit_heading :** *'UNIT'* IDENTIFIER *';'* ; **IDENTIFIER :** *[A-Za-z_][A-Za-z_0-9]*;* . . . . .

Figure 2: Grammar *MDL* e *ObjectPascal*

2.3 ObjectPascal Language

ObjectPascal is the Delphi enviroment Component-Based development language [11]. The *Visual Component Library* (VCL) [12] is a hierarchy of classes, written in *ObjectPascal*, that supports the applications development through the reuse of components and Object Inspector, using the latter for inspection and definition of objects.

Figure 3 illustrates the main screens of Delphi, where the windows with Tool Bar, Components Page, Object Inspector and Form Designer and its Source Code stand out.



```
unit Unit1

interface

uses
  Windows, Messages, SysUtils,
  Dialogs;
```

Figure 3: Delphi Enviroment.

3 *DEVELOPMENT OF FRAMECARDIO*

FrameCardio was developed (development *for* reuse) in four steps**: Defining Problem Domain, Specifying Components, Designing Components and Implementing Components**, according to Figure 4.

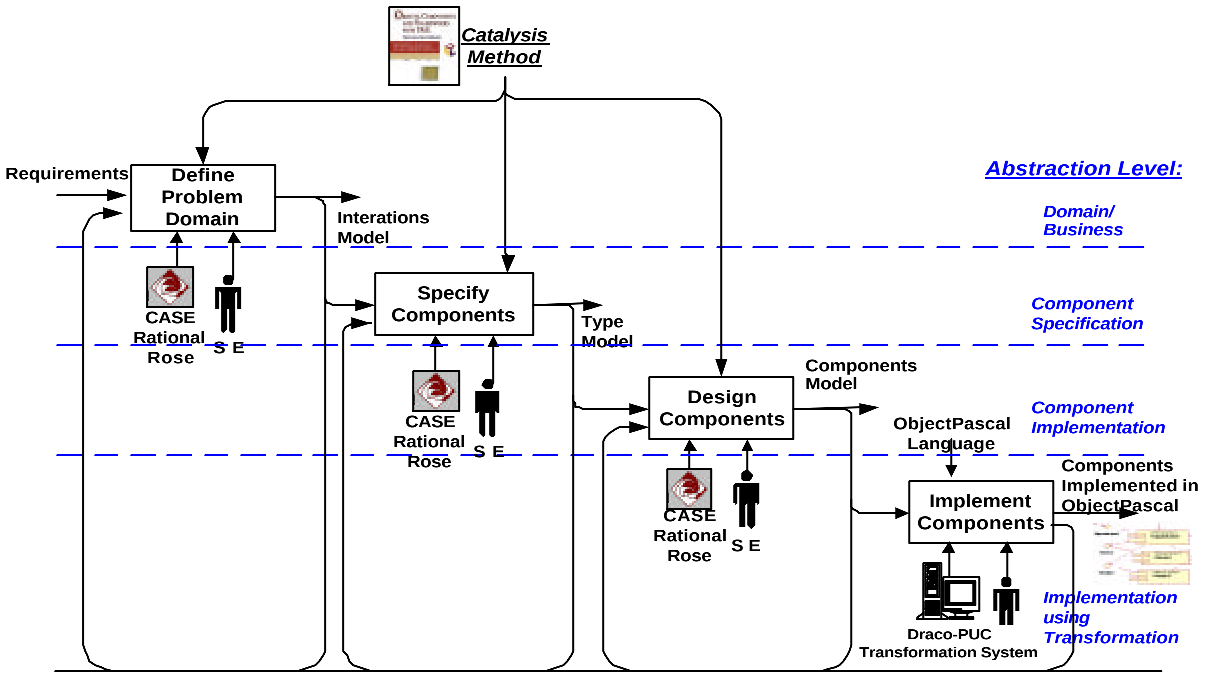


Figure 4: *FrameCardio* Development Process.

It begins with the common requirements of the Cardiology domain, to build components to be reused by the applications of this domain. The horizontal lines (blue) separate the steps of the development process, according to the levels Abstraction of the Catalysis Method***: Problem Domain, Components Specification, Components Internal Design and Components Implementation*** using Transformations, shown to the right of Figure 4.

3.1 Define Problem Domain

In the Defining Problem Domain step the Problem Domain terminology, the business process understanding, the actors' roles and the collaborations to specify the behavior of objects group of the cardiology domain have been defined. The main interactions models used in this step were the Use Cases, Actions and Collaborations. The identified requirements are specified initially by the Businesses Rules that state the Problem Domain understanding, represented in a Collaboration Model, including associations and use cases, reflecting the existent processes. Figure 5 illustrates a framework Collaborations Model with the actors: Doctor, Patient and Employee and the actions: *Accomplish Appointment, Accomplish Exam, Emit Decision and Accomplish Surgery.*

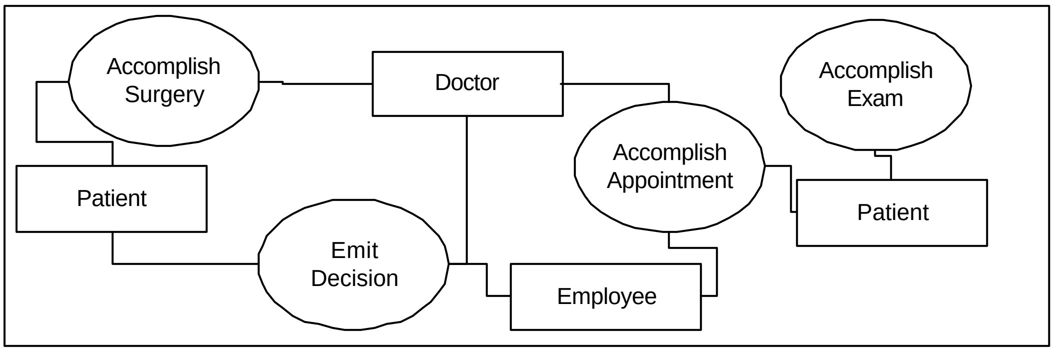


Figure 5 – Collaboration Model

Professionals and doctors of the cardiology area and a software system of the Heart Institute of Marília (ICM – Instituto do Coração de Marília-SP) were the main sources for requirements identification. The ICM software was developed and implanted by one of the authors that has been accomplishing its maintenance since 1997.

Meetings, interviews, legacy systems studies and observations of the system use sceneries have been used in the requirements identification. With the system in use, new requirements came along and, as time passed by, updatings have been made necessary to follow the technology changes. All these experiences were important to know the Cardiology domain, facilitating the Framework development.

Through the Collaboration Model refinement, in search of a better understanding of the framework functionalities, the main actions of the system are defined specifying the use cases. The use cases represent the actors' use sceneries.

Figure 6 illustrates the Use Cases Model, obtained from the Collaborations Model refinement of Figure 5. Actions can be modified or added, as for instance, the use cases registerPatient, registerDoctor, registerAppointment and sendAppointment, identified starting from the Accomplish Appointment action.

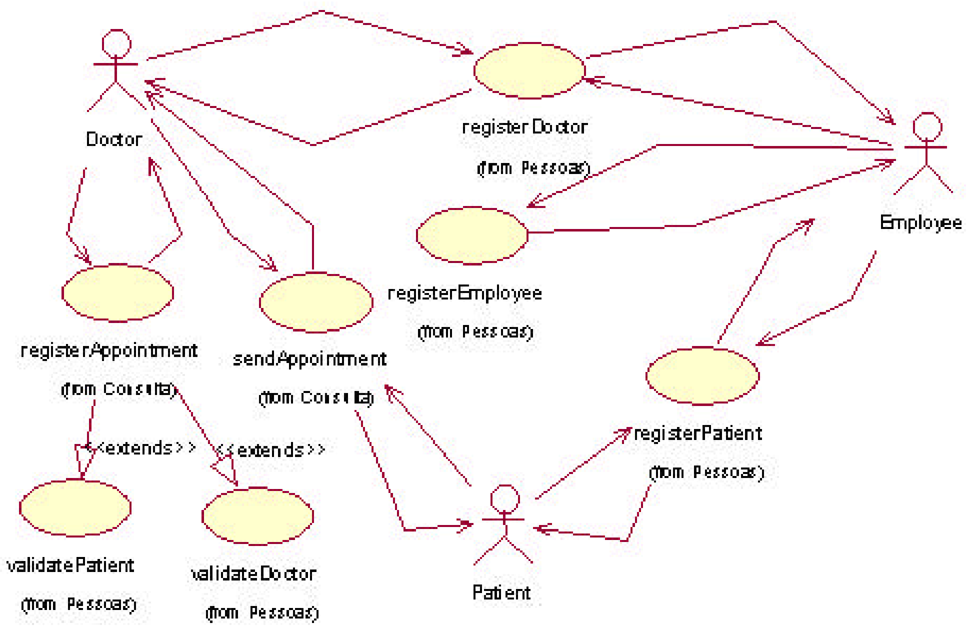


Figura 6 – Use Case Model

3.2 Specify Components

In the second step, *Specifying Components*, the components external behaviors were defined, with their responsibilities, operations and interfaces.

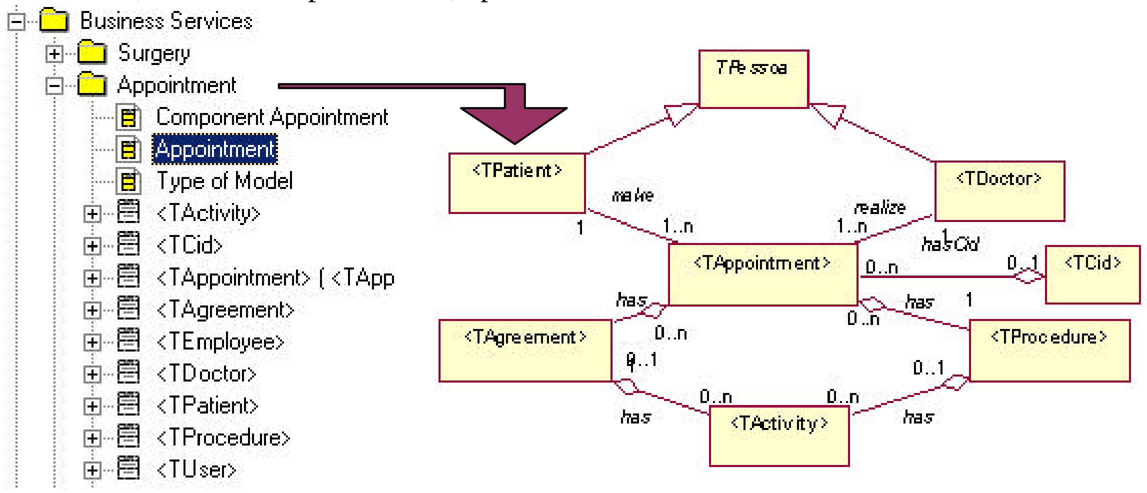


Figure 7 – Types Model

The main model is the Types Model, that describes the external behavior of an object, regardless of the implementation decisions. Figure 7 illustrares the Model Types obtained from the Use Cases Model refinement. The notation < > indicates the types that can be reused by FrameCardio applications.

The types are related through associations, aggregations or inheritance, with the respective cardinality, that states the minimum and maximum occurences of participant objects of a relationship. Another model specified in this step is the Interaction Model, represented in Sequence Diagrams witch detail the Use Cases behavior.

3.3 Design Components

Then, in the *Design Components* step, the Components Specified, in the Types Models and Sequence Diagram, are refined, considering the implementation technologies.

The Components Model represents the components physical architecture, with their interfaces for connection and their dependences, specified from the Classes Model refined of the Types Model. In this model we have the components interfaces, represented by little circles. An interface specifies the services accomplished by the component. A component can accomplish several interfaces, supplying a group of methods able to implement the services specified in the interface.

The connection among the components, or between a component that has access to the services of another component, occurs through the interface and is represented by a dependence relationship. Figure 8 illustrastes a Components Diagram, eliciting their interfaces and associations. In this case the *TAppointment* component is connected with the *TPatient* component through the *IPatient* interface.

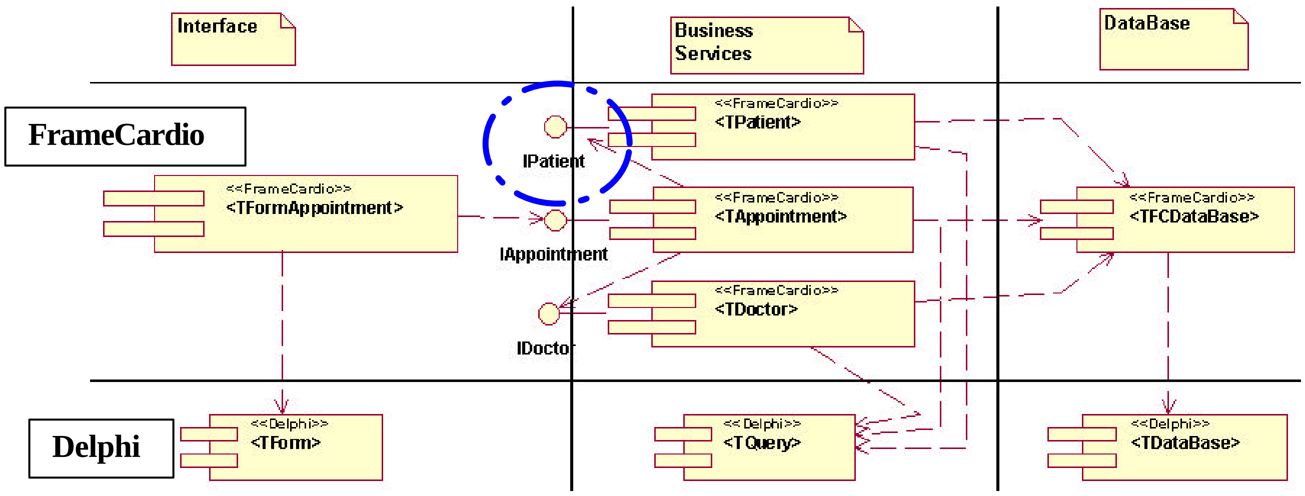

Interface
Business Services
DataBase
FrameCardio
<<FrameCardio>> <TPatient>
IPatient
<<FrameCardio>> <TFormAppointment>
<<FrameCardio>> <TAppointment>
IAppointment
<<FrameCardio>> <TFCDataBase>
<<FrameCardio>> <TDoctor>
IDoctor
Delphi
<<Delphi>> <TForm>
<< Delphi>> <TQuery>
<<Delphi>> <TDataBase>

Figura 8: Components Model –Appointment Package.

To facilitate the reuse of the 320 components of FrameCardio, they were organized in a Packages Model according to Figure 9. The packages were distributed in the following layers: ***User Services, Business Services and Data Services***.

The first layer, *User Services*, provides component for development of graphic and visual interfaces. The second layer, *Business Services*, provides components of the cardiology area, in three packages: **Appointment, Surgery and Decision**. These packages are reused by different Cardiology domain applications. The third layer, *Data Services*, provides access components to relational databases, that are used by the business rules.

Other available packages, as the one of **Delphi** components, can be reused, as the Model of Figure 9 shows.

The three-tier architecture makes the components reuse possible, increases the independence and facilitates the applications portability, that can be written in different languages, to access different databases managers and to use the same business rules.

This organization in tiers turns the system more flexible to support the technological changes without damaging its structure, increasing its life cycle.

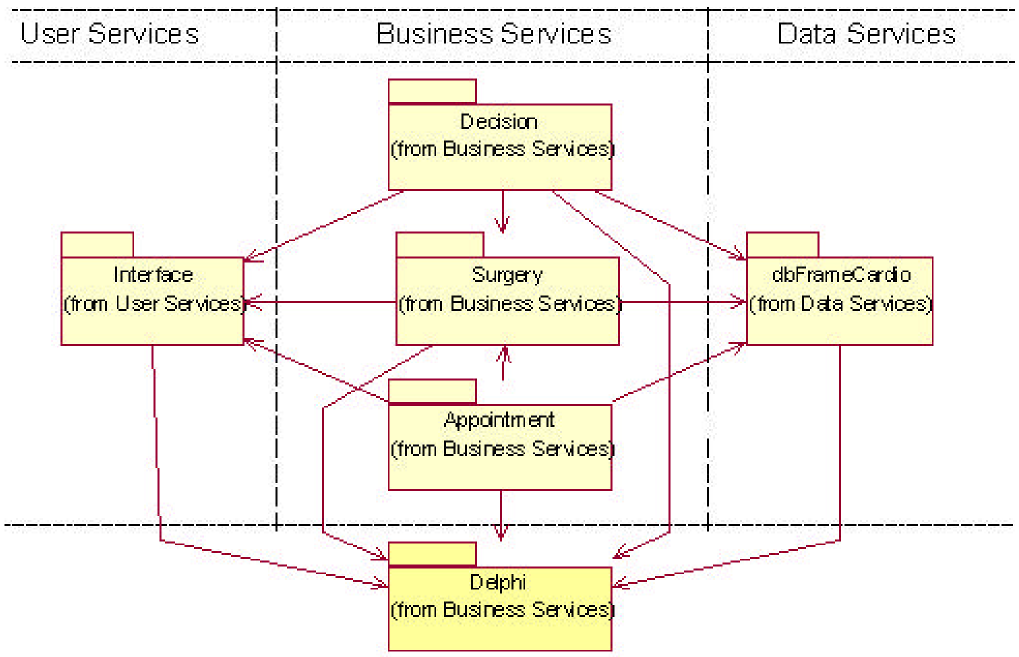
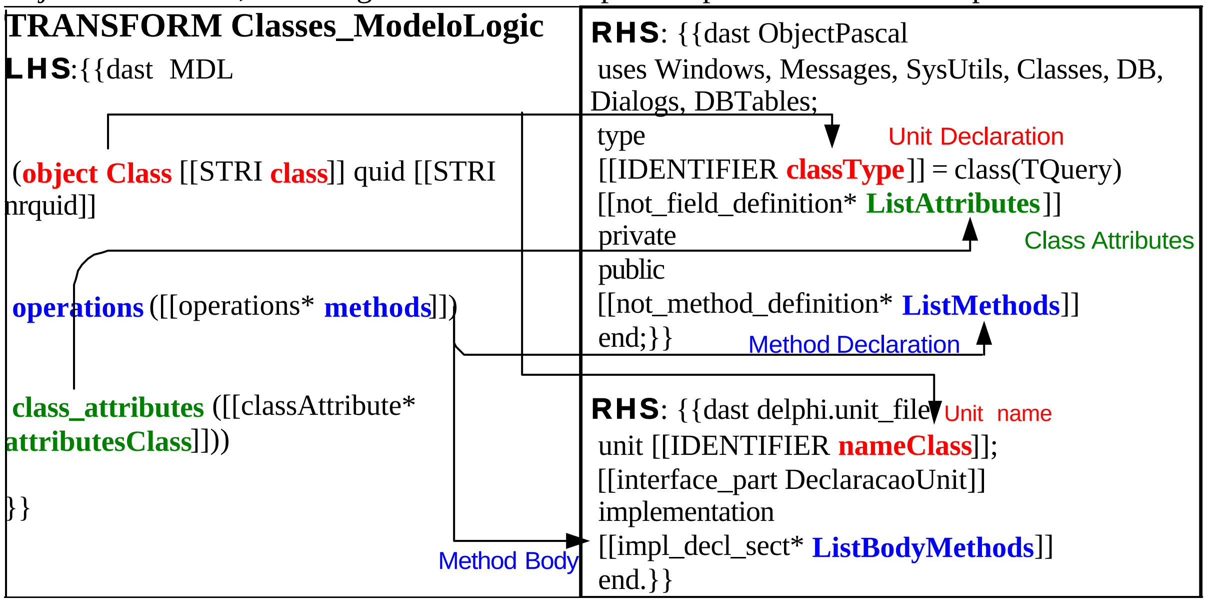


Figure 9: Framework *FrameCardio - Three-tier* Architecture.

3.4 Implement Components

The components are implemented based on their Internal Designs. The Draco-PUC Transformation Systems was used to generate *ObjectPascal* Code, from the components design MDL descriptions.

Figure 10 shows the *Transform 'Classes_ModeloLogic'*, that recognizes the specification of a class in *MDL*, in the control point LHS and generates the correspondent *ObjectPascal* code, according to the substitution pattern specified in the control point RHS.



```
TRANSFORM Classes_ModeloLogic
LHS:{{dast  MDL

(object Class [[STRI class]] quid [[STRI
nrquid]]

operations ([[operations* methods]])

class_attributes ([[classAttribute*
attributesClass]]))

}}
```

```
RHS: {{dast ObjectPascal
 uses Windows, Messages, SysUtils, Classes, DB,
Dialogs, DBTables;
 type
 [[IDENTIFIER classType]] = class(TQuery)
 [[not_field_definition* ListAttributes]]
 private
 public
 [[not_method_definition* ListMethods]]
 end;}}
```

```
RHS: {{dast delphi.unit_file
 unit [[IDENTIFIER nameClass]];
 [[interface_part DeclaracaoUnit]]
 implementation
 [[impl_decl_sect* ListBodyMethods]]
 end.}}
```

Figura 10: MdlToDelphi Transform

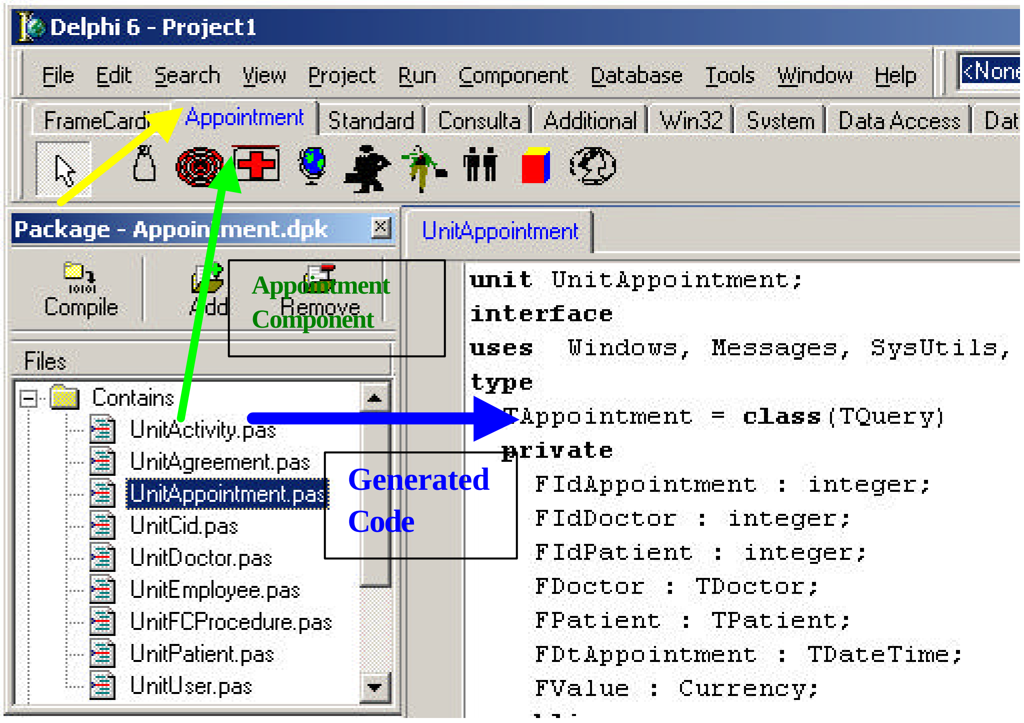
Figure 11 illustrates, to the left, the *TAppointment* Class MDL specification, and to the right, the correspondent ObjectPascal code generated. In this case, the specification *object Class* "TAppointment" created Type *TAppointment*, that implements a derived class of the TQuery class of Delphi. The specification object Operation *InsertAppointment* created the procedure *FCInsertAppointment*. The specifications *object ClassAttribute* IdAppointment and

DtAppointment created the attributes *FIdAppointment*, of the Integer type and *FDtAppointment* of the TDateTime type. The libraries of classes Windows, Messages and others are added according to the implementation needs.

MDL **Specification**	***ObjectPascal*** **Code Generated**
(object Class "<**TAppointment**>" quid "3C52D05602F7" operations (list Operations (object Operation "**Insert Appointment**" . . . . . class_attributes (list class_attribute_list (object ClassAttribute "**IdAppointment**" type "Integer") (object ClassAttribute "**DtAppointment**" type "Date")	unit UnitAppointment; interface uses Windows, Messages, SysUtils, Classes, Dialogs, DB, DBTables, UnitDoctor, UnitPatient; Type **TAppointment** = class(TQuery) F**IdAppointment** : Integer; F**DtAppointment** : TDateTime; procedure **FCInsertAppointment**; . . . . .

Figure 11: *ObjectPascal* Code Generated, from *MDL* specifications

A Figure 12 illustrates a generated code of a component with their interfaces.



```
unit UnitAppointment;
interface
uses  Windows, Messages, SysUtils,
type
  TAppointment = class(TQuery)
  private
    FIdAppointment : integer;
    FIdDoctor : integer;
    FIdPatient : integer;
    FDoctor : TDoctor;
    FPatient : TPatient;
    FDtAppointment : TDateTime;
    FValue : Currency;
```

Figure 12 –*TAppointment* Component implemented in *ObjectPascal*

3.5 Framework Life Cicle

The Software Development Processes presented follows the characteristics of the Spiral model of software development, in which the Software Engineer can return to the previous steps to refine the specified models and to obtain new Components Implementation. Next, a Case Study of the cardiology domain will be presented, reusing the *FrameCardio* components.

4 *CASE STUDY*

It is about a Medical Service System to the Patients in the Cardiology Clinic. The patient is assisted in the reception of the Clinic by an employee and led to the cardiologist's room. The appointment is accomplished and the results are registered in the database. The Doctor can verify if the Patient had already been registered in the system and update his information.

A presentation of each step of the development of this application is proceeded: **Model Application, Implement Application and Execute Application.**

4.1 Model Application

Initially, the Software Engineer, in the CASE tool, models the application according to the first level of Catalysis. The application requirements are specified, concerning about their business rules. Figure 13 illustrates the main application use cases, with an abbreviation description, their entrances and exits.

Nr	Desciption	Use Case	Entrance	Exits
01	Doctor accomplishes Appointment	accomplishAppoint	InfoApp	Msg01
02	Doctor verifies Appointments	generateAppoint	InfoGenerateApp	Msg02
03	Employee registers Doctor	registerDoctor	InfoDoctor	Msg03
04	Employee registers Patient	registerPatient	InfoPatient	Msg04

Figure 13: Use Cases - Application

The use cases are modeled, in Use Cases Diagrams that show the actors interacting with the system. Figure 14 illustrates, for instance, the doctor actor interacting to accomplish an appointment. In this level of the problem, the actors' relationships with the use cases are suitable sparing details that are not relevant for the system context.

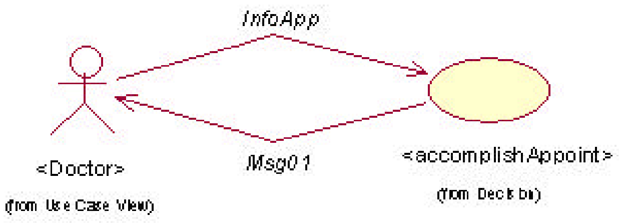


Figure 14: Use Case Model *accomplishAppoint*

In this first level, the Types Model of the application is still specified, concerning about "what " the application should do to assist their requirements. It is a model of high level of abstraction of the Problem Domain, where the essential types of the application are searched. Figure 15 illustrates a Types Model in this level of the development.

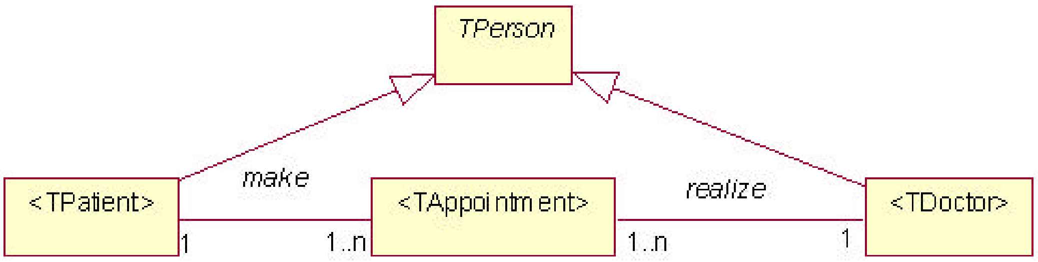


Figure 15: Model Types

Then, according to the second level of Catalysis, the models of the first level are refined, specifying the application components. In this level the Software Engineer concerns about the identification, behavior, and responsibilities of the components. Figure 16 illustrates the Application Model, obtained from use cases refinement, with the main types imported from FrameCardio. In this case, *Patient*, *Doctor* and *Appointment*, import the types <*TPatient*>, <*TDoctor*>, <*TAppointment*>, respectively from FrameCardio.

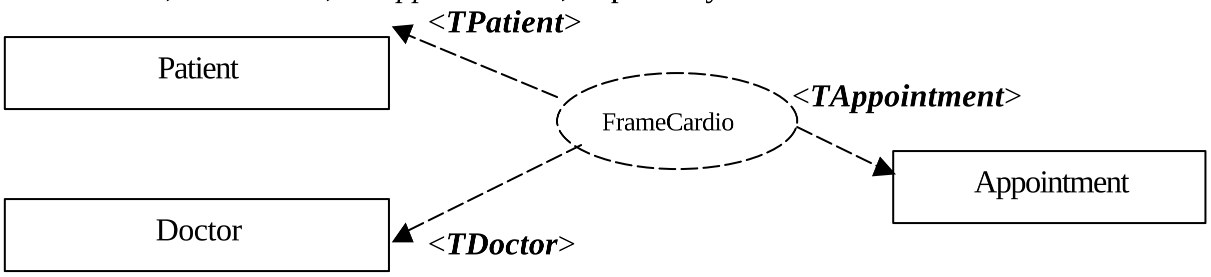


Figure 16: Application Types Model

The Types Model of the first level is refined in the Class Diagram Model that represents the application components, without concerning about their physical implementations. Figure 17 illustrates the new Class Model, where we can observe the new types *Doctor*, *Appointment* and *Patient*, derived from the FrameCardio types.

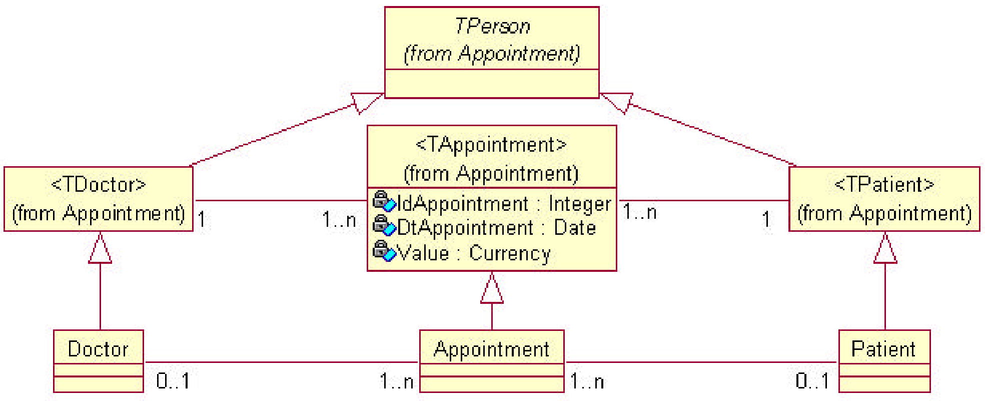


Figura 17: Application Class Diagram

Finally, in the third level of Catalysis, the Software Engineer specifies the Internal Designs of the Components, giving emphasis to their implementations and physical distributions. Figure 18 illustrates the Class Diagram of the *Appointment* component, obtained from the Types Model refinement, with its interface. In the *IAppointment* interface are the methods prototypes, implemented in the *TAppointment* component. <u>In this case the *validateAppointment*, *deleteAppointment*, *insertAppointment* and *selectAppointment* methods</u>.

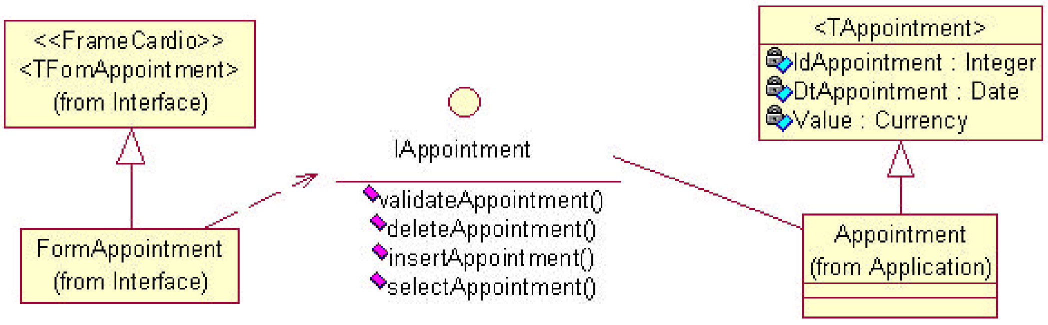


Figure 18 – Component Appointment Class Diagram

Figure 19 illustrates the application Components Model, where the components reused from FrameCardio, are indicated by the placeholders "<" and ">". We can observe that just the *FormAppointment*, *Appointment* and *DSAppointment* components, are specific of the application <<Application>>. The others are from FrameCardio or from Delphi itself. The stereotypes <<FrameCardio>> and <<Delphi>> illustrate the Framework components.

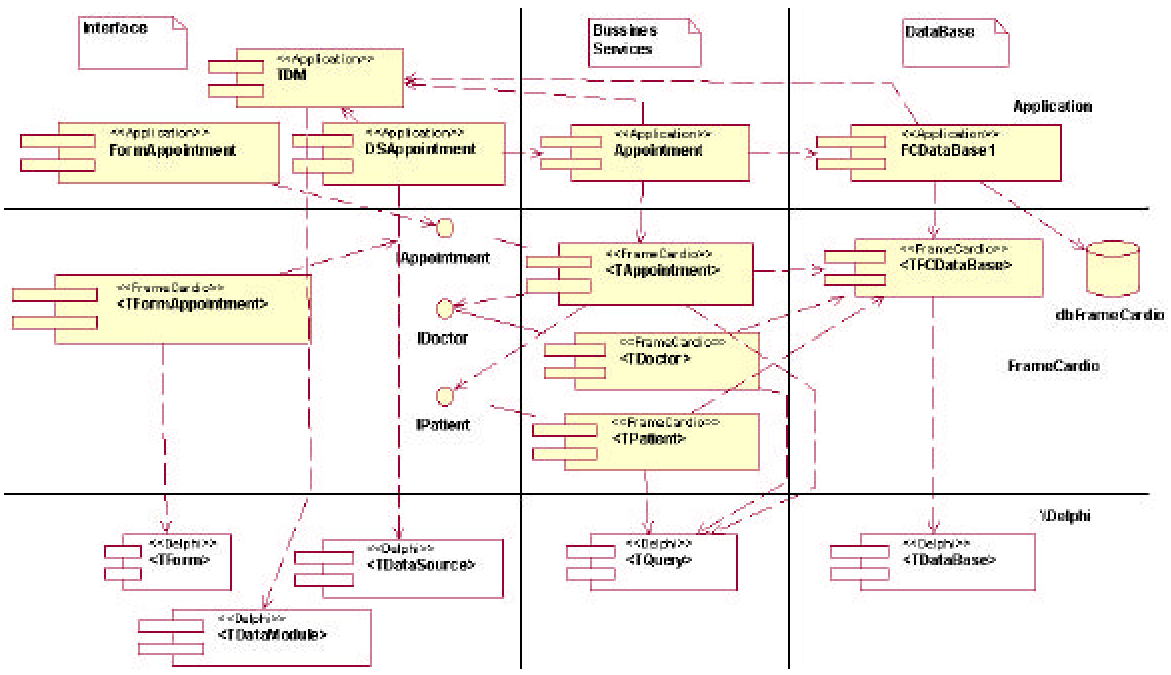

Interface
Bussines Services
DataBase
<<Application>> TDM
Application
<<Application>> FormAppointment
<<Application>> DSAppointment
<<Application>> Appointment
<<Application>> FCDataBase1
IAppointment
<<FrameCardio>> <TAppointment>
<<FrameCardio>> <TFCDataBase>
<<FrameCardio>> <TFormAppointment>
dbFrameCardio
IDoctor
<<FrameCardio>> <TDoctor>
FrameCardio
IPatient
<<FrameCardio>> <TPatient>
\Delphi
<<Delphi>> <TForm>
<<Delphi>> <TDataSource>
<<Delphi>> <TQuery>
<<Delphi>> <TDataBase>
<<Delphi>> <TDataModule>

Figure 19: Application Component Model.

4.2 Implement Application

In this step, the application is implemented in a component-oriented language. In this case, the application specifications, stored in a MDL file, are transformed, by Draco-PUC, into the ObjectPascal language. The code is generated based on the Classes Models of the application components. Interface components, generate another file, with .DFM extension, containing the definitions of the graphic part of the interface.

Figure 20 illustrates, to the left, part of an unitDM.pas, with the ObjectPascal code, of the Appointment component of the application and, to the right, the correspondent Diagram that shows the reuse of the FrameCardio implemented components. The *Patient* and *Doctor* components were connected with the *Appointment* component, demonstrating the Components "*Plug-Ins*" principle of the Catalysis Method.

```
unit UnitDM;
interface
uses  SysUtils, Classes, DB,
DBTables, FCDatabase,
UnitDoctor, UnitPatient,
UnitAppointment;
type
  TDM = class(TDataModule)
   Patient: TPatient;
   Doctor: TDoctor;
   Appointment:
TAppointment;
   FCDatabase1:
TFCDatabase;
   DSPaciente: TDataSource;
   DSConsulta: TDataSource;
   DSMedico: TDataSource;
```

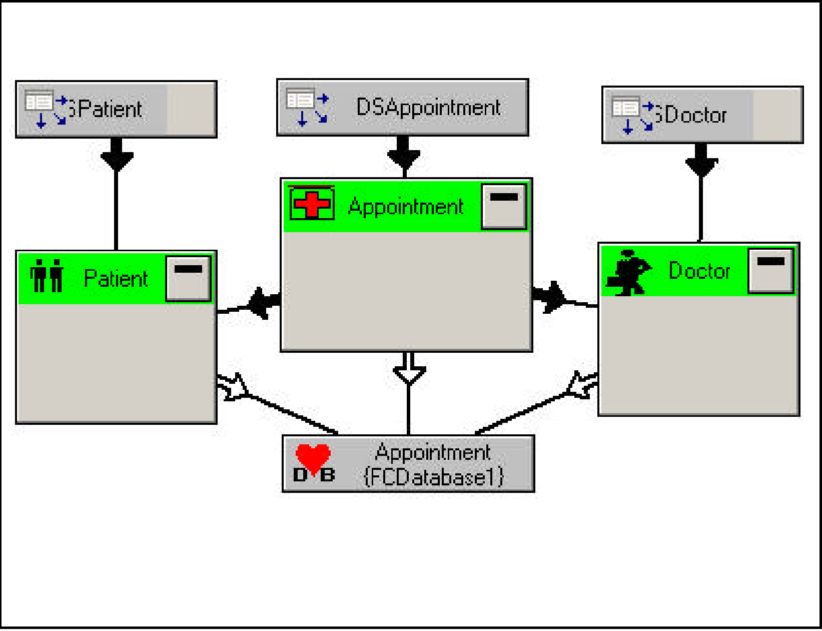
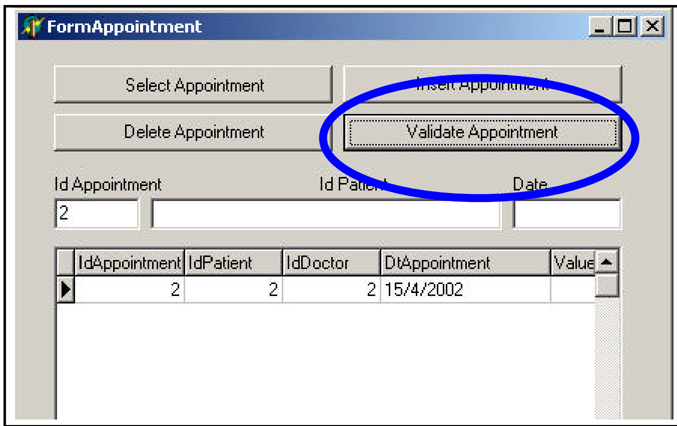


Figure 20: ObjectPascal Code Generated

4.3 Execute Application

In this step, the Software Engineer imports the generated code, in the *Delphi* enviroment, to execute it. In *Delphi*, the code is gathered in an application project to facilitate its management. The implementation in *ObjectPascal*, generated in the Implement System step, cannot be enough to assist all the requirements, mainly the non-funcional ones, related with the interface, safety, validation of data and access to database. Thus, the Software Engineer, using the visual resources of Delphi, can complement the project with other components that implement these requirements. The code generated by Draco-PUC, integrated to the code developed in Delphi, results in the implementation of the whole system. With the whole system implemented, it can finally be executed to verify if it suits to the specified requirements. In case of problems, or new requirements, the previous steps can be refferred to for corrections or additions of new requirements and, again, re-implement the system.

Figure 21 illustrates, to the left, an interface component generated and, to the right, FrameCardio FCValidateAppointment's method, being accessed by the IAppointment Interface of the Appointment Component.



Id Appointment: 2

IdAppointment	IdPatient	IdDoctor	DtAppointment	Value
2	2	2	15/4/2002	

```
procedure
TFormAppointment.ValidateAppointment(
Sender: TObject);
begin
DM. Appointment.FCIdAppointment
:= StrToInt(Edit1.Text);
DM.Appointment.FCValidateAppoint
ment;
end;
```

Figure 21: Application Execution

5 *CONCLUSION*

This paper presented Component-Based Software Development Processes that integrates different technologies, eliciting the Catalysis Method and a Transformation System, to develop a component framework of the Cardiology domain (FrameCardio). The main contribution of this work is to propose an Component-Based Software Development Processes, that integrates several mechanisms, guiding the software engineer as much in the development as in the reuse of components of a problem domain.

In the CASE tool, reusing the FrameCardio components, the Component-Based Application Design is obtained. The descriptions of the MDL specifications, that represent the design, are used to generate the code of the system in a Component-Based language. Draco-PUC Transformation System automated great part of the ObjectPascal code generation, of the framework and their applications.

Integrating an environment of visual programming, as Delphi, with the propose Software Development Processes, was possible execute the application to validate the specified requirements. In the final version it can be added, in the Delphi environment, components that treat the non-functional requirements, that had not been treated in the modeling of the system. A cycle of life of the software, that generates archetypes, facilitates the purification of the components, through successive refinements. The use of software components reveals each time more important to speed and to facilitate the Software Development Processes.

Due to the capacity of Draco-PUC Transformation System, of supporting different modeling domains and application, other languages, different from Catalysis and ObjectPascal, can be used in the proposed Software Development Processes. Although the process has been instanced for the Cardiology Domain and its applications, the authors believe that the steps of the presented processes and the integrated technologies to support it, can be used for the development of frameworks and applications of other domains of applications, particularly the similars.

Software Development Processes proposed give one more step in the automation of great part of the Software Engineer tasks, which can contribute in the reduction of time and of development costs of an application of the Cardiology Domain.

6 REFERENCES

[1] D'SOUZA, D.; WILLS, A. **Objects, Components and Frameworks with UML – The Catalysis Approach.** USA:Addison Wesley, 1998.

[2] BORLAND/INPRISE. *Object Pascal Reference*.

[3] LEITE, J.C.S., FREITAS, F.G., SANT'ANNA M. Draco-PUC Machine: A Technology Assembly for Domain Oriented Software Development**.** *3rd International Conference of Software Reuse***.** IEEE Computer Society Press. In proceedings, pp. 94-100. Rio de Janeiro. 1994.

[4] NEIGHBORS, J.M. The Draco approach to Constructing Software from Reusable Components. *IEEE Transactions on Software Engineering*. v.se-10, n.5, pp.564-574, September, 1984.

[5] RATIONAL SOFTWARE CORPORATION., http://www.rational.com/products/rose/prodinfo/index.jtmpl.

[6] COLEMAN, D. et al. **Object-Oriented Development – The Fusion Method**. Prentice Hall, 1994.

[7] FOWLER, M. **UML Destilled. Applying the Standard Object Modeling Language**. England:Addison Wesley, 1997.

[8] BOOCH, G. et al. **The Unified Modeling Language – User Guide**. USA: Addison Wesley, 1999.

[9] FUKUDA, Ana Paula. **Refinamento Automático de Sistemas Orientados a Objetos Distribuídos**. Dissertação de Mestrado. UFSCar, 2000

[10] PRESSMAN, R. S. **Engenharia de Software**. Makron Books: São Paulo, 1995.

[11] BORLAND/INPRISE. *Programming with Delphi 2001*.

[12] BORLAND/INPRISE. *Visual Component Library Reference*.