

LABORATOIRE



INFORMATIQUE, SIGNAUX ET SYSTÈMES
DE SOPHIA ANTIPOLIS
UMR 6070

ADAPTING ANALYSIS AND DESIGN TO SOFTWARE CONTEXT: THE JECKO APPROACH

Isabelle MIRBEL, Violaine de RIVIERES (Amadeus sas)

Projet MECOSI

Rapport de recherche
I3S/RR-2002-14-FR

avril 2002

RÉSUMÉ :

Des technologies orientées objet ont été développées pour appréhender la complexité liée au développement de nouveaux systèmes d'information. Elles proposent des solutions au travers de mécanismes et de techniques couvrant un large panorama de domaine d'application et d'applications. Mais les développements peuvent être très différents les uns des autres et une technique, une notation ou un mécanisme peut être utilisé différemment en fonction de l'application à développer. Une certaine flexibilité est donc nécessaire dans la méthodologie associée au développement en fonction du contexte de l'application.

Nous proposons JECKO, une approche flexible de l'analyse et de la conception où la flexibilité est proposée au travers d'un mécanisme de fragmentation. Pour adapter les activités d'analyse et de conception, des fragments sont sélectionnés en fonction du contexte de l'application, qui est spécifié au travers de quatre critères de base prédéfinis. Les fragments choisis constituent un itinéraire construit pour l'application à développer au travers du processus d'analyse et de conception. En suivant cet itinéraire, les activités d'analyse et de conceptions sont adaptées aux spécificités de l'application.

Dans cet article, nous présentons les différents fragments qui constituent le coeur de JECKO. Nous montrons comment le mécanisme de fragmentation permet de se focaliser sur les aspects critiques du développement dans le but de mieux appréhender sa complexité.

MOTS CLÉS :

analyse et conception orientées objet, méthodologie, flexibilité

ABSTRACT:

New object-oriented technologies have been developed in order to manage complexity inherent in new information system development. They propose solutions through rich sets of mechanisms and techniques, covering a large set of application domains and softwares. But developments are very different one from the others, and a given technique, notation or mechanism is used differently depending on the software under consideration. Flexibility is required from the methodology with regards to the software context.

We propose JECKO, a flexible approach to analysis and design where flexibility is handled through a fragmentation mechanism. In order to adapt the analysis and design activities, fragments are selected with regards to software context, specified through four predefined basic criteria. The chosen fragments constitute the route map built for the software being considered. By following this dedicated route map, the analysis and design activities are tailored for the software specificity.

In this paper, we present the different fragments JECKO framework is made of; and we show how our fragmentation mechanism allows to focus on the critical aspects of the development in order to better handle its complexity.

KEY WORDS :

object-oriented analysis and design, methodology, flexibility

Adapting Analysis and Design to Software Context: the JECKO Approach

Isabelle MIRBEL¹ and Violaine de RIVIERES²

¹ Laboratoire I3S

Route des Lucioles - BP 121
06903 Sophia Antipolis Cedex
France

Isabelle.Mirbel@unice.fr

² Amadeus sas

485 Route du Pin Montard, B.P. 69
06902 Sophia Antipolis Cedex
France

vrebuffel@amadeus.net

Abstract. New object-oriented technologies have been developed in order to manage complexity inherent in new information system development. They propose solutions through rich sets of mechanisms and techniques, covering a large set of application domains and softwares. But developments are very different one from the others, and a given technique, notation or mechanism is used differently depending on the software under consideration. Flexibility is required from the methodology with regards to the software context.

We propose JECKO, a flexible approach to analysis and design where flexibility is handled through a fragmentation mechanism. In order to adapt the analysis and design activities, fragments are selected with regards to software context, specified through four predefined basic criteria. The chosen fragments constitute the route map built for the software being considered. By following this dedicated route map, the analysis and design activities are tailored for the software specificity.

In this paper, we present the different fragments JECKO framework is made of; and we show how our fragmentation mechanism allows to focus on the critical aspects of the development in order to better handle its complexity.

1 Introduction

New object-oriented technologies have been developed in order to manage complexity inherent to new information system development [2]. New programming languages, middleware technologies, databases and modeling languages have emerged to cope with these complex developments. They try to propose solutions to handle such a complexity through rich sets of mechanisms and techniques. Adapted methodologies have also been developed to support and take

advantage of these techniques and mechanisms [11, 12, 9, 4]. Moreover, developments are very different one from the others, and a given technique, notation or mechanism may be used in a different way depending on the development under consideration. Simple methodology that has proved its power for certain types of software development may be quite unsuitable for others. There is no universal methodology [14]. Therefore it seems to be a need for flexible processes, adaptable to different kinds of software development. The need for situation-specific approaches, to better satisfy particular situation requirements, has already been emphasized [7, 10, 8]. Flexibility is required from the methodology with regards to the development context.

In this paper, we propose JECKO, a flexible approach to analysis and design: Flexibility is handled through a **fragmentation** mechanism. In order to adapt the analysis and design activities, **fragments** are selected with regards to software context, specified through four predefined basic criteria: (i) dealing with a running software (ii) dealing with a graphical user interface (iii) dealing with a database (iv) dealing with a distributed software.

The backbone of the JECKO **framework** is the **fragment**. A fragment embodied modeling rules and guidelines to help through the analysis and design activities (A&D-Ac). Guidelines are expressed with the UML notation [1] which is a well-known standard.

The chosen **fragments** constitute the route map built for the software being considered. By following this dedicated route map, the A&D-Ac are tailored for the software specificity.

In the following, after introducing the criteria describing the software context in Section 2, we describe the JECKO **framework** in Section 3. In Section 4 the different analysis and design phases and their associated fragments are described. Finally, in Section 5 we conclude.

2 The Software Context

Software development success requires initially the advised selection of modeling concepts (model elements, diagrams, artifacts, etc.) in order to deal efficiently with A&D-Ac. In this section, we present four criterias to specify the software context. Thanks to these criterias, the Analysis and Design Process (A&D-Pr) can be better adapted by selecting tasks to do and artifacts to use for getting accurate outcomes.

2.1 Dealing with Running Software

Most of the softwares are now built on top of running software. In this context, it is required to specify precisely what has to be kept from what has to be replaced or enhanced. To handle correctly such preservation, the A&D-Ac has to focus on the interfaces with the exiting software; and boundaries between both software pieces, new and running one, have to be precisely identified and documented. A dedicated A&D-Pr is therefore suitable.

Different aspects of running software may be taken into consideration. In addition to the code itself, the expertise about the functional domain (taken from the functionalities, data and screen shots) may also be of interest. The interfaces describing the relationships the running software has with other systems (softwares, databases, ...) should also be taken into consideration. These three aspects have to be distinguished because a given project could focus only on one aspect. For instance, if the project starts from Rapid Application Development (RAD), only functional domain aspect should be kept from the running software. Keeping the code is different from keeping the functionalities and/or from keeping the interfaces. In consequence, the A&D-Ac will be different with regards to the preservation kind. Different diagrams and modeling concepts may be more or less suitable to highlight different reuse aspects. Preservation will also have an impact on the forthcoming phases of the Software Process Development. It is important to early distinguish these three aspects (functional domain, interface, code) in the A&D-Pr.

The analysis activity is dedicated to clearly identify *what* has to be preserved, while the design activity focuses on *how* such preservation can be handled. To carry on properly the design activity, how the running software will be taken into consideration has to be specified precisely:

- it can be taken as it is; for instance, while interfacing a running component without any possibility to modify it (this is the case of bought component);
- or it can be slightly modified; for instance, while interfacing a running component, developed by the company, but used by other company softwares;
- or finally, it can be widely modified; for instance, when software evolves into new version and the development team is the owner of the running part.

The three preservation aspects (functionalities, interfaces, code) should be modulated in order to be better exploited during the development phase. We qualify each of them by : (i) *strong* when no modification is allowed, (ii) *medium* when modifications are allowed inside given boundaries, and (iii) *weak* when modifications are allowed with few limits. Preservation specification and qualification help to tailor the A&D-Pr to manage it more efficiently. In [13], the authors propose a profile to deal with such context through the UML notation.

2.2 Dealing with Graphical User Interface (GUI)

Lot of tools and techniques are currently available on the market and propose generic frameworks to facilitate GUI development, reducing the design activity for this aspect. In consequence, to fully benefit from such tools and techniques, the A&D-Pr has to be adapted by isolating the GUI specification from the business one; this is not so easy for end-users who are usually enclined to describe the software business only through its GUI. Therefore, an adaptation of the A&D-Ac to facilitate this distinction between the GUI and the business enhances A&D-Pr.

2.3 Dealing with Database

Dealing with database requires to organize data in such way that specific database rules (as the normal form decomposition) can be followed. Traditional A&D-Pr [5, 3] propose efficient ways for dealing with database design. The A&D-Pr should take advantage of such traditional techniques.

Moreover, dedicated concepts are required to built effective database: primary key, index, etc. To fully manage these concepts, additional information has to be captured through the analysis activity.

And finally, if the project under development includes also the choice of one database management system, the analysis outcome must be one key input for this choice.

For these three reasons, the A&D-Pr has to be adapted with regards to database aspect.

2.4 Dealing with Distributed Software

Distributed softwares are more complex to apprehend. Most of the time, they require particular organization of the A&D-Ac to fully handle the distribution aspects. Moreover, specific problems and requirements may be encountered with distributed software (link feature, speed, network security aspects, ...). The A&D-Pr has to be adapted to handle correctly such particularities.

The four criterias we have presented are simple and concrete. They only require an answer by *yes* or *no* to allow the A&D-Pr adaptation and to improve its outcomes. The software context is evaluated *a priori* with regards to these criterias.

3 The JECKO Framework

In order to be used by anyone without fundamentally changing his main A&D-Pr phases, the JECKO framework supports a decomposition into standard phases.

3.1 Facing Problems through Different Viewpoints: JECKO Phases

The **Requirement Analysis** deals with the requirement formalization. It covers explicit requirements, expressed by the end-user, as well as implicit ones, deduced by the analyst. Functional requirements, as well as technical ones, are carefully considered. Their pertinent organization facilitates the progress of the forthcoming phases of the A&D-Pr.

Then, the **Domain and Business Object Analysis** focuses on the specification of the business covered by the software to be developed. This is a crucial point of the analysis activity, in order to get a software fulfilling the requirements previously captured and also to anticipate future enhancements not yet fully identified.

Requirement Analysis and *Domain and Business Object Analysis* may be processed in parallel. They provide two complementary and essential view points on the software to be developed.

The third phase concentrates on the **Software and System Architecture**. Softwares are more and more complex from the functional and technical points of view. Most of the time, they involve heterogeneous environments, which increase considerably the complexity of the architecture to be deployed. Therefore, part of the A&D-Ac has to be dedicated to the specification of the software and the system architecture.

Then, the fourth phase, **Component Specification**, copes with the integration of the different components mainly through the identification of the interfaces among them. Potential reuse aspects are handled through this phase of the A&D-Pr.

Software and System Architecture and *Component Specification* phases may be processed in parallel. They provide two complementary and essential view points on the software to be developed.

Finally, through the *Internal Design* phase, the specification of each element (mainly component) previously defined is refined to allow the implementation.

3.2 Organizing Analysis and Design Work: JECKO Steps

Each phase reassemble related A&D-Ac dedicated to a given aspect of the A&D-Pr. In order to help in dealing with A&D-Ac, the different phases have been decomposed into **steps**. A step allows to better organize the activities associated with the current phase of the A&D-Pr. Four main steps come to view whatever is the phase under consideration.

- The first step, **Introduction** step, copes with preliminary tasks to enhance the forthcoming step progress.
- The second step, **Core** step, encompasses the main tasks of the current phase. These tasks concentrate on the elaboration of the phase outcome.
- The third step, **Refinement** step, focuses on organizing and refining the various diagrams and concepts built through the previous steps.
- The last step, **Investigation** step, concerns specific tasks required to better handle additional aspects of A&D-Ac and which appear to be crucial for the A&D-Pr success.

By organizing A&D-Ac into steps, we better distinguish the different kinds of activities one has to deal with when working on analysis and design. Moreover, we provide the same decomposition in each phase to help in having a coherent and systematic approach, as it is shown in Figure 1.

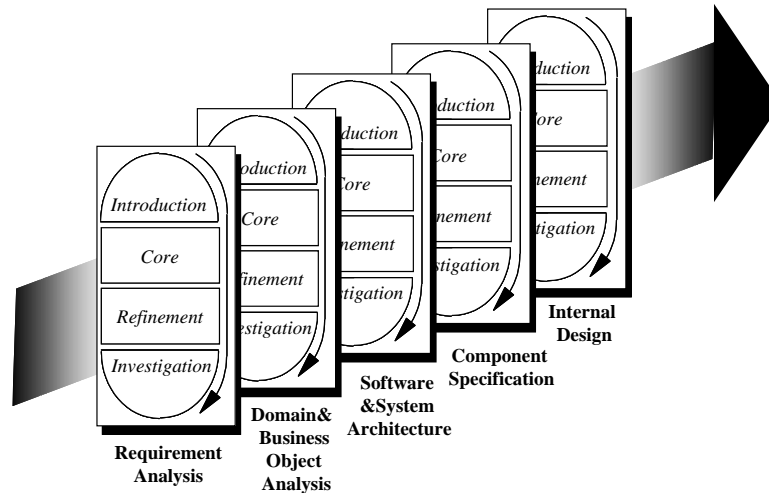


Fig. 1. The JECKO Framework

3.3 JECKO Backbone: the Fragment

The JECKO methodology proposes in each step a set of **guidelines**. In this paper, guidelines are expressed with the UML notation [6, 15, 1] which is a standard. To help in applying the guidelines, they are provided with additional information in what we call a **fragment**.

The originality of our work consists in the fragmentation proposed to handle flexibility; and especially the way fragments are organized to facilitate customization of any A&D-Pr with regards to the software context.

A fragment is defined with the following elements:

- **Name:** to identify the fragment;
- **Situation:** to position the fragment with regard to the software context;
- **Intention:** to present the fragment purpose;
- **Guidelines:** to explain how to proceed for answering the intention. In this paper, guidelines are given with the UML notation, but they may as well be expressed in another formalism.
- **Associated fragments:** to point out related fragments which have to be considered when defining the A&D-Pr. They share some aspects with the current fragment.

Whatever is the software context, A&D-Ac include unavoidable tasks: specification of the system boundaries, business description, etc; but they have to be handled slightly differently depending on the software context. Furthermore, context-related additional tasks may also be suitable through the A&D-Pr. Therefore, in JECKO, we distinguish *context-dependent* fragments from *context-independent* ones. They are respectively named **dedicated fragments** and **prime fragments**.

Figure 2 shows a fragment example, *Interleaving*, which belongs to the *Core* step of the *Domain & Business Object Analysis* phase: it is required for a development dealing with a running software, and more precisely when preservation deals with the code one. Related fragments, coming from the phases defined in JECKO, *Requirement Analysis*, *Domain and Business Object Analysis* (current one), *Software & System Architecture*, *Component Specification*, help to follow a coherent and systematic approach through the whole A&D-Pr. The *Intention* summarizes the objectives of the guidelines presented in the fragment; it includes the UML diagrams to be used.

Domain&BusinessObjectAnalysis::Core::Interleaving	
Situation	<input checked="" type="checkbox"/> Code preservation
Associated Fragments	RequirementAnalysis::Refinement::Interleaving, RequirementAnalysis::Investigation::Processing-Order, Domain&BusinessObjectAnalysis::Refinement::Interleaving, Software&SystemArchitecture::Refinement::Interleaving, ComponentSpecification::Refinement::Interfacing-1, ComponentSpecification::Refinement::Interfacing-2
Intention	► To stress what are the existing information and the new ones
Guidelines	UML diagram: Class diagram <ul style="list-style-type: none"> ► Stereotype attributes and operation should clearly distinguish what is existent from what is new. ► Stereotype new elements with <<new>>. ► With regards to attributes and operations that are classified as existent: <ul style="list-style-type: none"> ► If the preservation is <i>Strong</i>: take the element characteristics (type, length, etc.) from the running software and add them in the corresponding element. Use the <<re -use>> stereotype. ► If the preservation is <i>Weak</i> or <i>Medium</i>, choose to modify or not the element. To preserve it means to handle it as in the previous item; to modify it consists in stereotyping it <<to-be-modified>> and to keep from the running software only the information preserved (type, length, etc.). For operations, indicate precisely the modification to be done (as anote for example). ► Stereotype relationships among classes as done for attributes and operations. Characteristics to be kept have to be reported into the class diagram. Indicate possible modifications (cardinalities for instance) as anote.

Fig. 2. Fragment example

3.4 JECKO Framework Instantiation: the Route Map

By situating the software with regards to the four predefined criteria, the framework is tailored: suitable fragments (mandatory *prime fragment* and dedicated *specific fragment*) are selected to drive the A&D-Ac. These fragments, sequentially organized in time, constitute the *route map* associated with the software being considered. This is how, following this dedicated route map, the A&D-Ac is tailored for the software specificity.

The JECKO approach, which has been presented above, is summarized in Figure 3 using the UML notation.

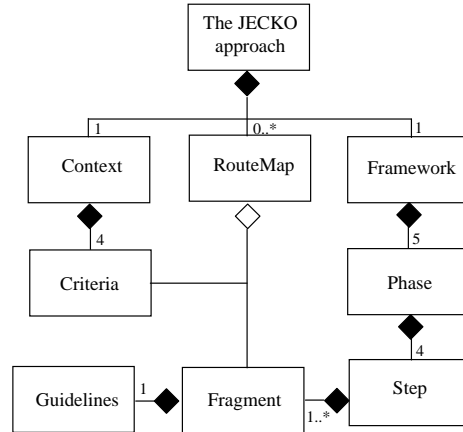


Fig. 3. The JECKO Framework

4 Handling Flexibility through the JECKO Approach

Flexibility is handled by selecting the fragments dedicated to the software under development. In this section we present the different fragments provided by each phase of our process.

4.1 Requirement Analysis

The fragments related to *Requirement Analysis* are presented in figure 4. They are detailed in the following.

Prime fragments: The *Requirement Analysis* focuses on the specification of the services provided by the software under development. The **Functionalities** fragment helps through this task. The **Decomposition** fragment helps in decomposing the functionalities into packages or area to better handle the requirements.

Then, an effort has to be made to organize the functionalities in order to help in clarifying the objectives of the development, its limitations and interactions with both users and systems. Furthermore, by checking if the services have been fully studied and specified, it allows to better control the forthcoming phases; this is the purpose of the **Organization** fragment. Then, the **Sequencing** fragment gives guidelines to provide additional diagrams in order to complete the description given through the use-case diagrams; especially when the number of

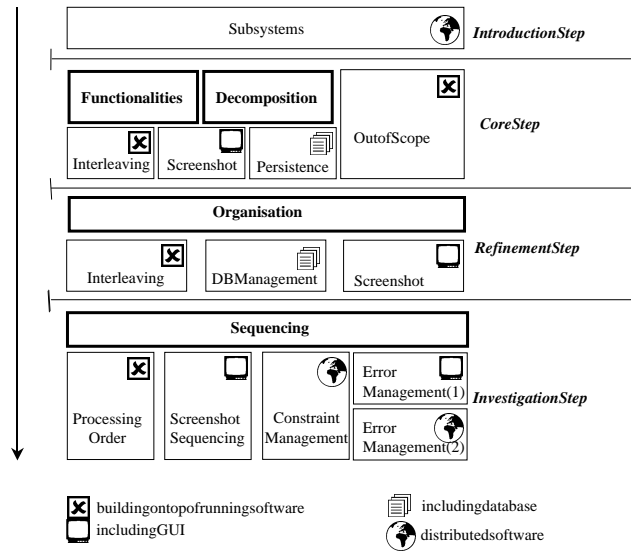


Fig. 4. The JECKO Framework - Requirement Analysis

services to be supported is high, it is interesting and important to show how they are related one to each other.

When building on top of running software: As the *Requirement Analysis* focuses on the specification of the software boundaries, when *building on top of running software* it is required to document very precisely interleaving between the running software elements to be kept from the new ones. In this context, the key point of requirement analysis is to clearly distinguish new functionalities from existing ones supported by the current running system, and a clear distinction has to be done between functionalities preserved as they are, and the ones enhanced by the new development. Therefore dedicated fragments, called *Interleaving*, are provided in the *Core* step and in the *Refinement* one to help the analyst respectively to distinguish these different kinds of functionalities and to organize them.

Moreover, functionalities of the running software, which are not directly related to the new development, may be of interest to help the understanding of what is or not provided through the new development. This is the purpose of the *Out-of-scope* fragment.

The usual way to describe functionalities (for instance through use-cases in the UML notation) focuses on the structural aspect. The dynamic point of view has also to be taken into account to indicate how the different functionalities may be chained together especially when a huge number of functionalities are supported by the software. Moreover, when distinguishing existing functionalities

from new ones, some of them are split through the *Core* and *Refinement* steps. It is therefore particularly suitable to then provide additional information to show how they may be chained together. This is the purpose of the **Processing-Order** fragment.

When software includes GUI: It is recommended to already think, through *Requirement Analysis*, about the screen shots associated with the software. This is the purpose of the **Screen-shot** and **Screen-shot-Sequencing** fragments. Functionality specifications are (i) completed with screen shot description, (ii) re-organized to help one to understand the associated screen shot, (iii) completed with additional diagrams to show the screen shot sequences.

Furthermore, error management has to be studied very carefully and already through this stage of the A&D-Pr in order to be homogeneous through the whole software. This is the purpose of the **Error Management** fragment.

When software includes database: Functionalities requiring persistence have to be specified to better anticipate the forthcoming phases of A&D-Pr, especially the *Software & System Architecture* one. Moreover, to clearly identify the functionalities where persistence is required may also help to check if database-related functionalities (as for instance, authentication, security purposes, etc.) have to be included in the scope of the software and have been fully handled. This is respectively the purpose of **Persistence** and **DB Management** fragments.

When software is distributed: *Distributed software* needs dedicated tasks to split the software into different physical entities (commonly named subsystem) before specifying the different functionalities. This is the purpose of the **Subsystems** fragment in the *Introduction* step. Then, constraints among the related subsystems have of course to be clearly established. **Constraint-Management** fragment handles this particular aspect. As it has already been highlighted when *software includes GUI*, error have to be dealt with in an homogeneous way, through **Error Management** fragment. Different fragments are provided for software including GUI and distributed software because they deal with different error kinds.

4.2 Domain and Business Object Analysis

The different fragments related to *Domain and Business Object Analysis* are presented in figure 5. They are detailed in the following.

Prime fragments: This phase of the A&D-Pr is dedicated to the study of the business objects, mainly supported by the **Business Object prime** fragment. Most of the time, objects are described in a static way through classes, attributes and relationship among classes. Sometimes it may be interesting to investigate

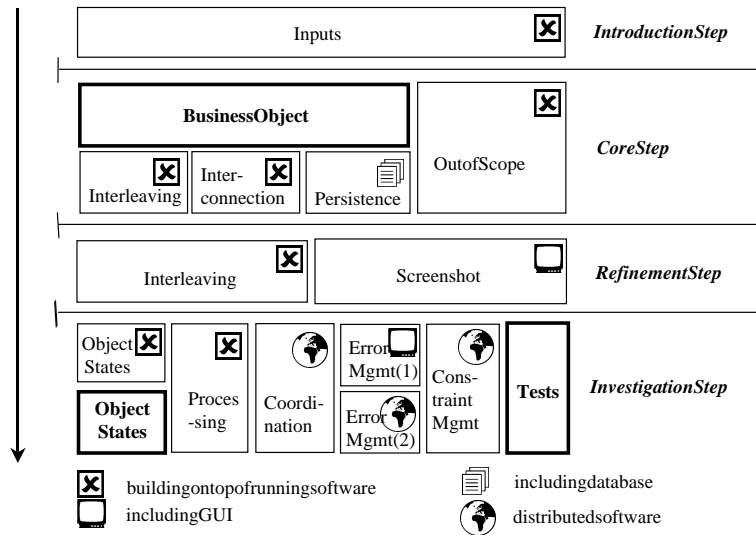


Fig. 5. The JECKO Framework - Domain and Business Object Analysis

the specification by a dynamic point of view. This is already given through the methods in the classes but it could be better described and related to the events through state diagrams. This is the purpose of the **Object State** fragment in the *Investigation* step.

Functional tests are handled through the **Tests** fragment.

When dealing with a running application: Information about the business object may have to be taken from the running part of the software. The **Inputs** fragment deals with such situation.

As it has been explained through the *Requirement Analysis* phase, it is crucial to document the interleaving between existing and new aspects of the software. It has also to be done for business object. This is the purpose of the **Interleaving** fragments in the *core* and *refinement* steps. In the same way, business objects from the running software, which are not directly related to the new development, may be of interest to help to understand what is or not provided through the new development. The **Out of scope** fragment helps in taking advantage of documenting these business objects.

Communications with the running software are represented in terms of actors interacting with the software under development, as explained through the *Requirement Analysis* phase. Actors may represent constraints on the way the new software will be working. Therefore, interactions have to be described very precisely. This is the purpose of the **Interconnection** fragment.

The description of the business objects specified through the development may need to be completed by a dynamic point of view through state diagrams, as it has been explained above. To be coherent, if business object from the running software are statically described through the A&D-Pr, they also have to be described from a dynamic point of view, for the same reasons. This is the purpose of the **Object states** fragment.

When developing on top of an existing software, some of the services provided by the whole software may indeed be supported by both the new and the existing part of the software. In this case, it is important to explain what is the process flow associated with a "shared" service, what are the interactions between the new and the running parts of the software to let one understand how the software is working. If only part of a service has to be developed in the new software, it may also be interesting to document the activities related to the running part of the service, to let one understand how the new part has to be developed and how it will be related to the running one. For these kind of task, a dedicated fragment is proposed: **Processing** fragment.

When software includes GUI: The screen shots elaborated through the *Requirement Analysis* phase have to be refined through the *Domain and Business Object Analysis* with regards to the object specification which is now more complete. This is the purpose of the **Screen shoot** fragment. In the same way, errors which have been used through the functionality specification have to be summarized in a class diagram. It will ensure coherence and consistency in the way errors will be managed by the software. This is the purpose of the **Error Management** fragment.

When software includes database: Business objects requiring persistence have to be distinguished: it is useful to handle the information required to support persistence in the forthcoming phases of A&D-Pr. Information about candidate key for instance have to be already given. This is the purpose of the **Persistence** fragment.

When software is distributed: *Distributed software* is made of different subsystems communicating together and described through the *Requirement Analysis* phase. At this stage of the process, it is important to refine the specification of the information (messages, signals) exchanged by the different subsystems with regards to the business objects which has been described in the current phase. Therefore, a dedicated fragment, **Coordination**, is proposed to help dealing with such a specification.

In *Distributed software* constraints among the related subsystems have to be clearly established, as it has already been enlightened through the *Requirement Analysis* phase. Constraint and error specification may be refined and better described through hierarchies of classes. The **Constraint-Management** fragment handles this particular aspect.

As it has already been highlighted when *software includes GUI*, errors have to be dealt with in an homogeneous way, through the **Error Management** fragment. Different *Error Management* fragments are provided for *software including GUI* and *distributed software* because they deal with different error kinds.

4.3 System and Software Architecture

The fragments related to *System & Software Architecture* are presented in figure 6. They are detailed in the following.

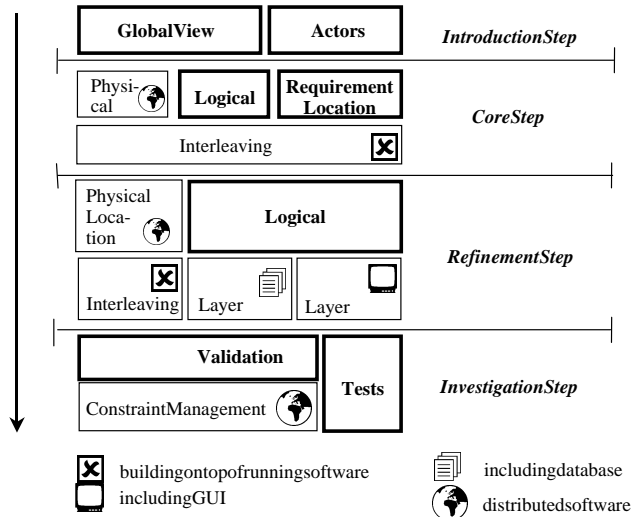


Fig. 6. The JECKO Framework - System & Software Architecture

Prime fragments: The *Core* part of the *System and Software Architecture* deals with the decomposition of the software into logical components. The **Logical** fragment helps through this task.

The components encapsulate the business object specified in the previous phase and realize the requirements expressed through the first phase of the process. Therefore, they have to be coherent with what have been specified through the *Requirement Analysis* phase. The **Requirement location** fragments helps in associating requirement specifications to components.

But before dealing with these two main tasks, it is also important to provide a global view of the application to explain how it will be working. It is especially meaningful if the software is part of a set of softwares collaborating together. The **Global View** fragment gives indication to deal with such documentation.

Moreover, specification with regards to relationships with actors have to be refined. Through the `Actors` fragment, it is explained how to better deal with this aspect of the specification. One has, for instance, to define if the information is provided by or to the actor or the format used to exchange information, because they are constraints to be taken into account through the specification.

Then, at the end of the phase, it may also be helpful, as it was done through the *introduction* step in the *Global View* fragment to show how the software will be working and to explain it through the use of the different components. This is the purpose of the `Validation` fragment.

Integration testing is handled through the `Tests` fragment.

When dealing with a running application, as it has been explained through the previous phases of the process, it is important to specify precisely the relationships among the running and the new parts of the software have to be specified precisely. With regards to the architecture, the existent and new components have to be distinguished and their relationships with the actors clearly identified. This is the purpose of the *interleaving* fragments in the *Core* and *Refinement* steps.

When software includes GUI as well as when software includes database, components dealing with the business have to be separated from the components related to the GUI and the components dealing with the database. GUI components are isolated to allow to add or remove GUI without modifying the business. Business rules are also isolated from the persistence, not to be dependent on persistence means. The `Layer` fragments help in following this n-tiers architecture decomposition.

When software is distributed, the physical software architecture has to be specified in addition to the logical one presented in the *prime* fragments. Dedicated fragments, named `Physical` and `Physical Location`, are proposed. Additional information may be given with regards to the technical constraints underlying the technical feasibility of the software (kind of relationship between the components, required features: secure connexion, ...). The `Constraint Management` fragment helps in dealing with these aspects of the specification.

4.4 Component Specification

The fragments related to *Component Specification* are presented in figure 7. Fragments are detailed in the following.

Prime fragments: A software includes various kinds of interfaces: public and private ones, provided by actors or to them, constrained or not. Each of them has to be specified with regards to specific constraints. It is therefore important to

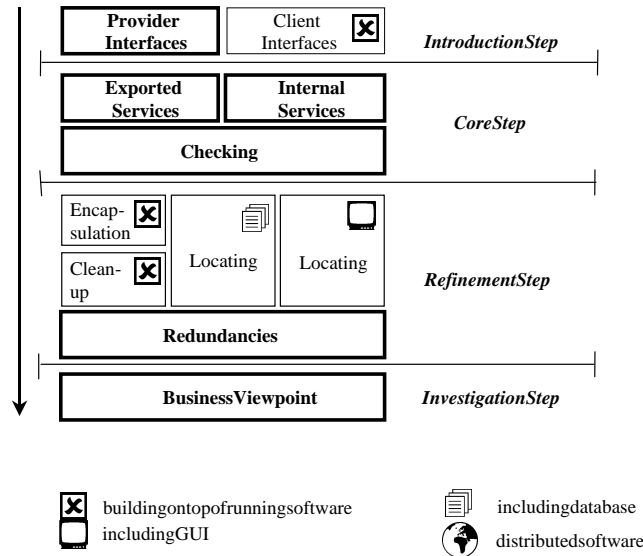


Fig. 7. The JECKO Framework - Component Specification

clarify the different kinds of interfaces the software has to deal with, in order to specify each of them in the good way, taking into account the right constraints.

We distinguished interfaces provided by the actor we are collaborating with from interfaces dealing with interactions among the different components our software is made of, detailed through the **Internal Services** fragment. About collaboration with the actors, we distinguish the actors from which we are dependent and deal with these kind of specification in the **Provider Interfaces** fragment. Actors to which we provide services are then studied through the **Exported Services** fragment.

Then, the **Checking** fragment helps in apprehending the whole set of interfaces specified through the current phase and to check it is complete and coherent.

Then, through the *refinement* step it is possible to reorganize the interfaces specified through the previous step through merges, transfers or splits, using the **Redundancies** fragment, to get an homogeneous interface specification for the software.

And finally, as this phase of the A&D-Pr deals with a very precise point of the specification (the definition of the interface) it may also be of interest to look at the software from another point of view. Instead of looking at each component and trying to describe its interfaces, one could look at the different services provided by the software from the end-user point of view (i.e. the functionalities given in the *Requirement Analysis* phase) to ensure it is fully supported

through the chosen architecture. Such a view point is presented in the **Business Viewpoint** fragment.

When dealing with a running application, in addition to the constraints taken from the actors providing services to our software (specified through the *prime Provider Interfaces* fragment) actors already collaborating with the running part of the software and for which our software is provider of information have to be studied. This is done through the **Client Interfaces** fragment. Then, services slightly modified have to be studied very carefully and one has to decide where to implement the modification: in the running component, in a new component or in a component dedicated to the modification. The **Encapsulation** fragment deals with this kind of problem. This is particularly useful when dealing with a legacy application.

Also when dealing with running components or dependent actors, it may be interesting to look at the existing interfaces and try to check what is still used and what is usefulness. This is the purpose of the **Clean-up** fragment.

When software includes GUI, as well as **when software includes database** manipulation rules are associated with the information. Display format for GUI, key unicity for databases are examples of manipulation rules associated with the business but also with the underlying technology. In these specific case, one has to decide where the manipulation rule has to be placed: in the GUI component, in the database component or in the business component. The **Locating** fragments deal with this aspect of the *Component Specification* phase.

4.5 Internal Design

The fragments related to *Internal Design* are presented in figure 8. Fragments are detailed in the following.

Prime fragments: After decomposing the software into components and specifying the different interfaces provided and used by the different components, the internal design aims at specifying more in detail the core of each component. This is the purpose of the **Realization** fragment. It is completed by the **Tests** fragment, in the *Investigation* step to handle the unit testing.

When dealing with a running application, only the components that will be developed have to be specified. Therefore a **Selection** fragment is proposed in the *Introduction* Step. Moreover, after the tasks associated with the *Core* step have been processed, it is crucial to check that the software will be working properly and will answer to the services described through the *Requirement Analysis*. Of course, this has to be checked whatever is the software context. But *when dealing with a running application*, it is much more difficult because parts of it are already implemented and others are not. The **Completeness** fragment helps through this work.

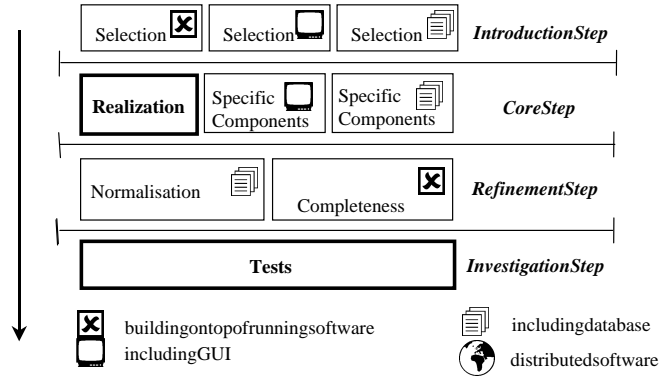


Fig. 8. The JECKO Framework - Internal Design

When software includes GUI, it is made of GUI components and business components. There exist dedicated technologies to deal with GUI, with facilities to implement them. Therefore, some of the GUI components may not need to be described as precisely as others and especially as the business components. Therefore, for *software including GUI*, the *Introduction* step includes a Selection fragment, and through the *Core* step, dedicated guidelines to deal with the design of GUI components are given in the Specific Component fragment.

For the same reasons, **when software includes database** similar fragments are proposed: Selection and Specific Component. Moreover, dealing with a relational (or object-relational) database implies to deduce a normalized data schema from the class diagram elaborated through the previous phases of the A&D-Pr. The Normalization fragment helps through this task.

5 Conclusion and Future Work

In this paper, we have presented JECKO, a flexible approach to analysis and design. A&D-Ac are adapted with regards to software *context*, specified through four predefined basic *criteria*: (i) dealing with a running software (ii) dealing with a GUI (iii) dealing with a database (iv) dealing with a distributed software. Thanks to this context, suitable fragments may be selected to better deal with the specificity of the software being considered. Furthermore, the fragments are sequentially organized into the JECKO framework. We distinguish *prime fragments* from *specific fragments*. A *prime fragment* is used whatever the software context might be, while *specific fragment* is associated with one of the four predefined criteria. The main purpose of a fragment is to propose modeling rules and guidelines to help through the A&D-Ac. The chosen *fragments*, ordered following the JECKO framework, constitute the *route map* built for the software.

Using this dedicated *route map*, the A&D-Ac are tailored to focus on the critical aspects of the development in order to better handle its complexity.

In the future, we would like to improve our fragments by integrating users feedback on our approach. Moreover, we would like to enrich the development context. In addition to the software criteria, information related to the project (i.e. time pressure, dependency with other projects) [10] may be taken into account.

And finally, we would like to enhance the JECKO framework by weighting the fragments with regards to the designer expertise to provide end-users adapted route maps. Depending on the person in charge of the A&D-Ac (junior or senior analyst, business or technical one) the A&D-Pr may be slightly different.

References

1. Jacobson I. Booch G., Rumbaugh J. *The Unified Modeling Language User Guide*. Object Technology Series. Addison-Wesley, 1998.
2. C. Rosenthal-Sabroux C Cauvet. *Ingnierie des systmes d'information*. Number 2-7462-0219-0. Herms, 2001.
3. E.F. Codd. Further normalization of the database relational model. In '*DataBase Systems*', Rustin(ed), Prentice-Hall publishers. 1972.
4. D. D'Souza. *Catalysis: Objects, Components, and Frameworks with UML*. Object Technology Series. Addison-Wesley, 1998.
5. A. Flory and J. Kouloumdjian. A model and a method for logical database design. In S. Bing Yao, editor, *Very large databases: Fourth International Conference on Very Large DataBases, West Berlin, Germany, September 13-15, 1978*, pages 333-341, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1978. IEEE Computer Society Press.
6. Object Management Group. The UML notation. <http://www.omg.org/>.
7. R.L. Glass I. Vessey. Applications-based methodologies. *Information System Management*, pages 53-57, Fall 1994.
8. C. Rolland J. Ralyte. An assembly process model for method engineering. In M.C. Norrie K.R. Dittrich, A. Geppert, editor, *Advanced Information Systems Engineering*, number 2068 in LNCS, pages 267-283. Springer, June 2001.
9. R. Lecat J.L. Cavarero. *La Conception orientee objet, evidence ou fatalite*. Ellipses, 2000.
10. B. Hodes K. van Slooten. Characterizing IS development projects. In R.J. Welke S. Brinkkemper, K. Lytinen, editor, *IFIP TC8, WG 8.1/8.2*, pages 29-44, August 1996.
11. P. Krutchen. *The Rational Unified Process*. Object Technology Series. Addison-Wesley, 2000.
12. A. Rochefeld M. Bouzeghoub. *OOM, la conception objet des systemes d'information*. Hermes Sciences, 2000.
13. I. Mirbel and V. de Rivieres. Towards a UML profile for building on top of an existing application. In *Information Resources Management Association International Conference*, Seattle, USA, May 2002.
14. J. Ralyte. *Ingenierie des methodes a base de composants*. PhD thesis, Universite Paris I - Sorbonne, January 2001.
15. Booch G. Rumbaugh J., Jacobson I. *The Unified Modeling Language Reference Manual*. Object Technology Series. Addison-Wesley, 1998.