JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2004**

Vol. 3, No. 8, September-October 2004

A UML based Framework Design Method

H. Ben-Abdallah, Faculté des Sciences Economiques et de Gestion de Sfax
N. Bouassida, **F. Gargouri**, **A. Ben-Hamadou**, Institut Supérieur d'Informatique et de Multimédia de Sfax, Tunisie

Abstract

Object-oriented frameworks offer reuse at a high design level promising several benefits to the development of complex systems. However, framework design remains a difficult task due to the generality and variability frameworks must encompass. In addition, traditional object-oriented design methods only deal with the design of specific applications and do not facilitate the design of frameworks.

In this paper, we present a UML-based framework design method called FBDM. The method offers a design language, called F-UML, and a semi-automatic design process both of which supported by a CASE environment. The design language F-UML visually distinguishes among the fixed components and the adaptable components of a framework. The design process for F-UML is based on stepwise, bottom-up unification rules that apply a set of comparison criteria on various applications in the framework domain. The design method is illustrated and evaluated through the design of a framework for electronic commerce brokers.

1 INTRODUCTION

Frameworks promise increased productivity, shorter development times and a higher quality of applications since they allow developers to reuse their previous experience in problem solving both at the design and code levels [Johnson 1988]. In fact, they are an extension to object-oriented designs, that helps reuse at a level of abstraction higher than classes, patterns [Gamma 1995] and components.

An object-oriented framework represents a software architecture that captures several applications' behaviours in a particular domain. It is composed of a set of concrete and abstract classes with their relations. It is organized in two parts [Pree 1994] : a *core* (also called *frozen-spot*) that is common to all applications derived from the framework, and *hot-spots* that represent the variable parts which allow a framework to be adapted to a particular application. Schmid [Schmid 1997] defines two types of hot-spots:

whitebox hot-spots adapted by implementing some of their methods and classes, and *blackbox hot-spots* adapted through composition.

Traditional object-oriented design methods (e.g., OMT [Rumbaugh 1991]) deal with the design of only specific applications. They are inappropriate for the design of frameworks since they lack concepts to determine and express hot-spots, an essential concept that should be clearly expressed in order to avoid any framework misuse. This deficiency motivated several proposals of new framework notations (c.f., [Riehle 2000], [Fontoura 2000b], [Sanada 2002]) and design processes (c.f., [Schmid 1997], [Koskimies 1995], [Fontoura 2000a]).

In this paper, we present a framework design method that offers a UML-based design language called F-UML [Bouassida 2001] and a bottom-up design process [Bouassida 2002]. F-UML is an UML profile [Bouassida 2003b] that increases the expressiveness of UML by adding tags and graphical annotations to UML use cases, class, pattern and sequence diagrams. The extensions help to distinguish visually between the core of the framework and its hot-spots and guide the user in instantiating a framework. The design process is based on a bottom-up strategy that generates a framework design by unifying a set of application designs. It is composed of three main steps: The first step extracts the domain specifications and potential uses of the framework through unification of the use case diagrams of different applications in the domain. The second step models the framework static features through unification of the class diagrams of the given applications. The last step extracts the dynamic framework features through the unification of sequence diagrams.

The FBDM design method is supported by a toolset [Ayadi 2003] that allows an easy representation of syntactically, well-defined frameworks [Bouassida 2003b] and a semi-automatic generation of a framework design. The toolset prompts the framework designer only to decide on the completeness of certain relations (inheritance, association,….), which requires a deep domain knowledge.

This paper is organized as follows. Section 2 overviews existing framework design methods. Section 3 and 4 present the framework design language F-UML and its design process, respectively. Section 5 illustrates the method through the design of a framework for electronic commerce brokers. Section 6 evaluates the method. Finally, section 7 summarizes the paper and outlines future work.

2 OVERVIEW OF FRAMEWORK DESIGN METHODS

In this section, we first outline a set of concepts necessary in a framework design language. We then use the outlined concepts to examine current framework design languages that are based on UML. Finally, we present current framework design processes.

The following five requirements detail out concepts necessary in a design language for frameworks [Bouassida 2001]:

1. The framework design notation must provide for a means to describe statically the framework:
    a) classes and their relations (association, generalization, aggregation);
    b) core; and
    c) whitebox and blackbox hot-spots.
2. Within a whitebox hot-spot, the notation must statically guide the user to the potential changes they are expected to introduce. For example, the notation indicates that the user is expected to redefine the code of a method, or the user may add inheriting classes, etc. This criterion facilitates a correct reuse of a framework.
3. The notation must contain concepts for regulating the interactions within a framework by:
    a) explicitly showing the collaborations between objects instantiated from the framework classes;
    b) clarifying the object responsibilities, contexts on which the responsibilities depend and how the objects may combine the different responsibilities; and
    c) being abstract and independent of unessential implementation details that may unnecessarily tie the design to a specific environment and limit the framework generality.
4. The notation must show the framework aim and potential uses of the, i.e., it must show scenarios of the framework instantiations.
5. The notation must be unambiguous to facilitate the correct comprehension of the framework.

Current Framework Design Languages

[Rational 2001] models a framework through three UML diagrams: a class diagram enriched with packages, a collaboration diagram and a use case diagram. The enriched UML class diagram expresses the static structure of a design (**1.a**). However, it does not distinguish between the classes in the core and those in the hot-spots (**1.b-c**). Although the name of an abstract class is in italic in the UML notation, this is not sufficient to deduce all the hot-spots. The UML collaboration diagram successfully shows the object interactions (**3.a**) and responsibilities (sender/receiver) (**3.b** partially). However, working at the message exchange level can be too detailed and does not indicate how and in which context the framework works. The UML use case diagram defines a set of external actors and their possible uses of the system. It could therefore be used to define the aim and possible contexts (**4**).

Fontoura et al. [Fontoura 2000b] propose a UML profile for frameworks, called UML-F, where a design is expressed by a class diagram and a sequence diagram both extended by *presentation tags* (e.g., complete, incomplete), *basic modeling tags* (e.g., fixed, application, framework) and *essential pattern tags* (e.g., FacM-Creator, FacM-ConcreteCreator). The added tags are used to mark, essentially, the complete and incomplete parts, the variable parts in the diagrams and the roles of diagram elements. In this notation, the extended class diagram represents the framework classes and relations (**1.a**). However, according to the tag definitions, this notation only identifies the whitebox hot-spots (**1.c** partially and **2**). In addition, several tags are complementary and thus redundant (e.g., *complete* and *incomplete*, *application* and *framework*). Furthermore, the combined pattern tags and presentation tags could overcharge the diagram and impede the understanding of the design. The extended sequence diagram guides the user when adapting framework interactions (**2** partially), and it explicitly shows the object collaborations (**3.a**) and responsibilities (**3.b** partially). However, similar to the UML sequence diagrams, it remains at a detailed level.

Sanada [Sanada 2002] presents an UML extension that aims to be comprehensive and well defined. However, most of the extensions proposed have already been defined by Fontoura[Fontoura 2000b], and the only difference is the constraint “*covariant*” which shows that adding a subclass to a certain class might result in adding a subclass to another one.

Riehle [Riehle 2000] proposes a role modeling language that adapts the OORAM methodology [Reenskaug 1996]. The proposed language represents a framework through a class model with an *extension-point class set* (points of extension), *a built-on class set* (framework interface) and *a free role type set* (the use of the framework by other frameworks). Overall, this notation represents the architecture and collaborations in a framework (**1.a** and **3.a-b**) and describes the framework context (**3.b**). However, it focuses more on framework composition than framework adaptation. For instance, it does not visually distinguish between extension-point classes and frozen classes in the framework. Therefore, one cannot easily recognize the whitebox and blackbox hot-spots.

Current Framework Design Processes

Current framework design processes can be classified as either bottom-up or top-down. Bottom-up design works well where a framework domain is already well understood, for example, after some initial evolutionary cycles. In this case, the design process starts from a set of existing applications and generalizes them to derive a framework design (c.f., [Koskimies 1995], [Schmid 1997]). On the other hand, top-down design is preferred when the domain has not yet been sufficiently explored. In this case, the design process starts from a domain analysis and then constructs the framework design (c.f., [Aksit 1998]).

Koskimies and Mossenback [Koskimies 1995] propose a two-phase bottom-up framework design process. The first phase, called *problem generalization*, generalizes a representative application in the framework domain into “the most general” form. In the second phase, called *framework design*, the generalization levels of the previous phase are considered in a reverse order leading to an implementation for each level. The implementation of the framework at level *i* requires adding specific classes and applying various design patterns on the framework. The last step in the design phase is to apply the resulting framework to the initial example problem of the generalization phase. This design process lacks guidelines for the problem generalization phase. In addition, both the reuse degree of the resulting framework and the ease of deriving the framework depend on how well the original application represents the domain. Furthermore, the resulting framework does not provide for reuse guidelines; that is, it does not clearly identify nor does it guide the designer in finding the framework core and hot-spots.

Schmid [Schmid 1997] decomposes the framework design process into three steps:

1. design of a class model for an (arbitrary) application in the framework domain;
2. analysis and specification of the domain variability and flexibility, i.e., identification of the hot-spots; and
3. generalization of the class model by applying a sequence of transformations that incorporate the domain variability.

This design process leaves it to the developer's expertise to identify the hot-spots during the second step.

Pree [Pree 1994] proposes a framework design process based on combining hot-spots specified as metapatterns. These latter are a set of design patterns that describe how to construct frameworks. This design process focuses on hot-spot combination without defining how to determine them.

Fontoura et al [Fontoura 2000a] propose a design process that considers a set of applications as viewpoints (i.e., perspectives) of the domain. The process informally defines a set of unification rules that describe how the viewpoints can be combined to compose a framework. The result of applying the unification rules is *a template hook model* that represents the hot-spots through template and hook methods. After developing the template hook model, the developer has to find which meta-pattern should be used to model each hot-spot. The resulting framework is an OMT class diagram that does not completely specify the framework; in particular, it neither distinguishes between the two hot-spot types, nor does it not specify the object interactions. In addition, this process does not address semantic issues in the unified applications (e.g., synonyms, homonyms,…); it supposes that all the semantic inconsistencies between the viewpoints have been solved beforehand.

3 THE DESIGN LANGUAGE F-UML

The development of the design language F-UML was motivated by the design criteria outlined earlier. In F-UML, a framework design consists of the following four UML based diagrams:

1. A use case diagram that determines the framework scope, objectives and domain limits (criteria **4**). The extensions to the use case diagram are summarized in Table 1.

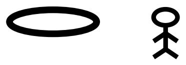
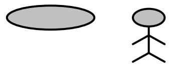
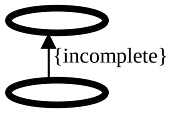
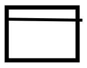
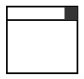
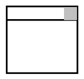
Graphical Notation	Explanation	Objective
	A highlighted use case (actor) border	to show the framework core
	A use case (actor) filled in gray	to show the framework hot-spot
	The constraint {incomplete} associated to the inheritance relation between use cases or actors	to show that it is possible to add an inheriting actor (use case) in an application reusing the framework.

Table 1. F-UML extensions to UML use case diagrams

2. A class diagram that describes the static architecture of a framework. The extensions to the class diagram (see Table 2) let the user distinguish between the core of the framework and its hot-spots (**1**) and guide him/her in adapting a whitebox hot-spot to a specific application (**2**).
3. A pattern diagram that shows the design patterns and metapatterns identifying the roles of the framework classes (**3.a-b**). The pattern diagram helps understanding the interactions among the objects of the framework.
4. Sequence diagrams that describe possible interactions between various object instances of the class diagram (**3.a, 4**). While the sequence diagrams might present too detailed information compared to the previous diagrams, they can provide important information in understanding how to use, and consequently reuse, a framework. The F-UML extensions to the sequence diagram are summarized in Table 3.

The F-UML diagrams can be enriched with OCL constraints [OCL 1997] to specify constraints and invariants on classes and types and pre and post conditions on methods.

The F-UML notation is a UML profile whose syntax (derived from the UML meta-model) has a formal semantics [Bouassida 2003b] (criteria **5**).

Graphical Notation	Explanation	Objective
	A highlighted class border	to show the framework core
	A square filled in black at the top-right corner of a class	to show the framework blackbox hot-spot
	A square filled in gray at the top-right corner of a class	to show that this class with all its inheriting classes are in the framework whitebox hot-spot

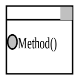
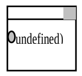
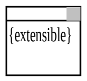
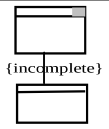
	A circle filled in gray in front of a method's name	to show a virtual method
	The tag *undefined* for a method with a varying signature	to show a method with an undefined signature
	The tag {extensible} in a class	to show that the class can be adapted by adding/removing attributes/methods
	The UML constraint *incomplete* on a relation (generalization, aggregation, association)	to show that the framework may be adapted by adding other related classes

Table 2. F-UML extensions to the UML class diagram

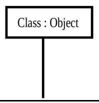
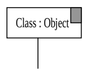
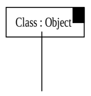
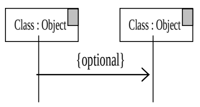
Graphical Notation	Explanation	Objective
	A highlighted object border	to show the framework core
	A square filled in gray at the top-right corner of an object	to show the framework whitebox hot-spot
	A square filled in gray at the top-right corner of an object	to show the framework blackbox hot-spot.
	The tag optional attached to a message as proposed by [Fontoura 2000]	to show that the message may not exist in an application adapting the framework

Table 3. F-UML extensions to UML sequence diagrams

4 THE FBDM DESIGN PROCESS

The F-UML bottom-up design process helps the framework designer to structure the framework by determining its core, blackbox and whitebox hot-spots. It starts from several application designs and goes through three unification steps (in fact, Roberts [Roberts 1996] states that three applications sufficiently represent their domain):

1. Design of the framework use case diagram: The unification process first extracts use cases common to all the applications and puts them as the framework core. Secondly, it extracts the different use cases and puts them as hot-spots.
2. Design of the framework class diagram: The unification process first extracts common classes and puts them as the framework core. Secondly, it puts the

remaining classes as hot-spots. For certain relations, the designer is probed to decide on their completeness.

3. Design of the framework sequence diagrams: The unification process identifies optional messages and tags them as hot-spots.

Currently, the pattern diagram is not obtained through unification. It is obtained from the class diagram (obtained in step 2) by filtering out the details inside the classes and by matching the resulting structure to a set of design patterns and metapatterns. At this level of representation, the user is interested in the roles and interactions between the various classes of the framework.

The above three unification steps use a set of unification rules based on semantic comparison criteria. These latter rely on linguistic definitions to define semantic equivalence, generalization-specialization, variation and composition between names.

In the remainder of the paper, to facilitate the presentation of unification process, we note:

- The class *C* (object *O)* in the application A_i as C_{Ai} $(O_{Ai};)$;
- The use case *U* (actor *A*) in the application A_i as U_{Ai} (A_{Ai});
- The message *M* in the application A_i as M_{Ai}; and
- The application A_1 as the application containing the minimal number of actors and use cases in the use case diagram (classes in the class diagram or messages in the sequence diagram).

Use case diagram unification

The unification of use case diagrams relies on semantic correspondences among the actors, use cases and their relations. These latter are expressed by the following relations:

- N_equiv(A_{A1},...,A_{An}) means that the names of the actors are either identical or synonym, e.g., Consumer$_{A1}$-Buyer$_{A2}$.
- N_var(A_{A1},...,A_{An}) means that the names of the actors are a variation of a concept, e.g., employee-contractual, employee-permanent, employee-vacationer.
- Gen_Spec(A_{A1};A_{A2},...,A_{An}) means that the name A_{A1} is a generalization of the specific names A_{A2} ,…, A_{An}, e.g., Person$_{A1}$-Employee$_{A2}$.
- N_dist(A_{A1},...,A_{An}) means that none of the above relations holds.

The relations between use cases are defined in a similar manner.

The design of the framework use case diagram is guided by the six rules depicted in Figure 1. As illustrated in this figure, the core and hot-spots of the framework are derived automatically. In addition, the designer’s intervention is guided (rule 2 and 3); it is needed to decide on the completeness of the diagram, which requires domain expertise.

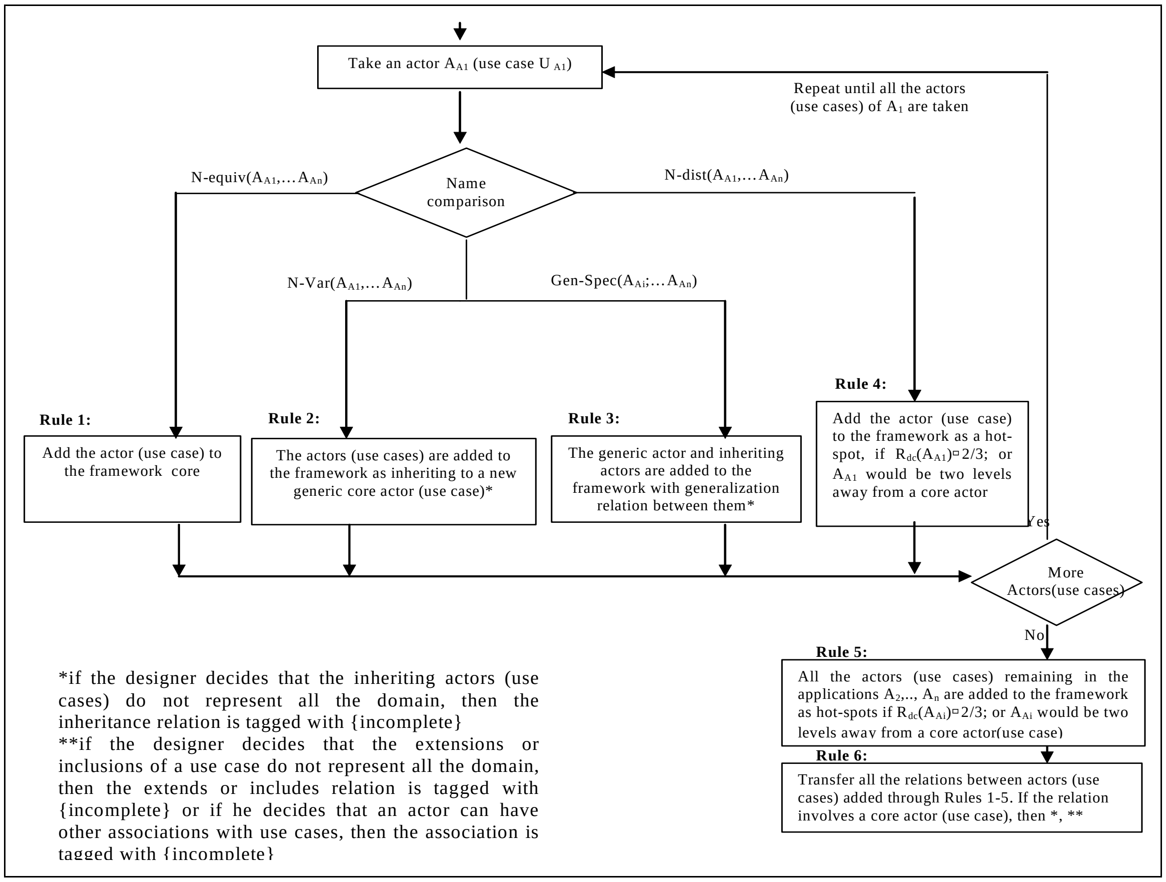

Take an actor A_{A1} (use case U_{A1})
Repeat until all the actors (use cases) of A_1 are taken
N-equiv(A_{A1},…A_{An})
Name comparison
N-dist(A_{A1},…A_{An})
N-Var(A_{A1},…A_{An})
Gen-Spec(A_{Ai};…A_{An})
Rule 1:
Add the actor (use case) to the framework core
Rule 2:
The actors (use cases) are added to the framework as inheriting to a new generic core actor (use case)*
Rule 3:
The generic actor and inheriting actors are added to the framework with generalization relation between them*
Rule 4:
Add the actor (use case) to the framework as a hot-spot, if $R_{dc}(A_{A1})$≥2/3; or A_{A1} would be two levels away from a core actor
Yes
More Actors(use cases)
No
Rule 5:
All the actors (use cases) remaining in the applications A_2,.., A_n are added to the framework as hot-spots if $R_{dc}(A_{Ai})$≥2/3; or A_{Ai} would be two levels away from a core actor(use case)
Rule 6:
Transfer all the relations between actors (use cases) added through Rules 1-5. If the relation involves a core actor (use case), then *, **
*if the designer decides that the inheriting actors (use cases) do not represent all the domain, then the inheritance relation is tagged with {incomplete}
**if the designer decides that the extensions or inclusions of a use case do not represent all the domain, then the extends or includes relation is tagged with {incomplete} or if he decides that an actor can have other associations with use cases, then the association is tagged with {incomplete}

Fig. 1: Design of the use case diagram

Rules 4 and 5 add or ignore hot-spot actors (use cases) according to the domain coverage ratio $R_{dc}(A_{Ai})$:

$$R_{dc}(A_{Ai}) = \frac{\text{number of occurrences of } A_{Ai} \text{ (or its variations or its equivalents) in } A_1,.., A_n}{\text{number of applications}}$$

Class diagram unification

The design of the framework class diagram consists of the eight unification rules shown in Figure 2. The unification rules essentially take classes “common” to all of the applications as the framework core and add the remaining (specific) classes as hot-spots. These rules use semantic correspondence criteria to compare class names, attributes and operations. The class name comparison criteria use five relations among class names. Four of these relations are defined in a way similar to the relations between actors. The fifth relation reflects class composition:

- N_comp(CA1;CA2,...,CAn) means that the name CA1 is a composite of the components CA2,…,CAn, e.g., HouseA1-RoomA2.

The attribute comparison criteria use four relationships to compare the attribute names and types:

- Att_equiv(CA1,..., CAn) means that the classes have eitheridentical or synonym attribute names with the same types.
- Att_int(CA1,..., CAn) means that the classes CA1,..., CAn have attributes in intersection.
- Att_conf(CA1,..., CAn) means that there exists at least one attribute of CA1 having a name equivalent to some attributes of CA2,..., CAn but these attribute types are different.
- Att_dist(CA1,..., CAn) means that none of the above relations holds.

The operation comparison criteria use four relations (Op_equiv(C_{A1},...,C_{An}), Op_int(C_{A1},...,C_{An}), Op_dist(C_{A1},...,C_{An}), Op_Conf(C_{A1},...,C_{An})) to compare the operation names and signatures (returned types and parameter types). These relations are defined in a way similar to the attribute comparison relations.

In the unification process shown in Figure 2, Rule *5* deals with the generalization-specialization relation in a manner similar to *N-Comp* relation of Rule *4*. Furthermore, Rule *3.b* may add new inheriting classes to the framework. The addition depends on the significance of the number of attributes and methods in an inheriting class *C* with respect to another class *C'*. This is defined using the ratio R_{sig}:

$$R_{sig}(C, C') = \frac{number\ of\ attributes\ and\ methods\ that\ belong\ to\ C\ and\ C'}{number\ of\ attributes\ of\ C + \ number\ of\ methods\ of\ C}$$

Informally, a class *C* has a significant number of attributes and methods with respect to a class *C'*, if R_{sig} is greater than a fixed threshold (e.g., 50%) that can be fixed by the framework designer.

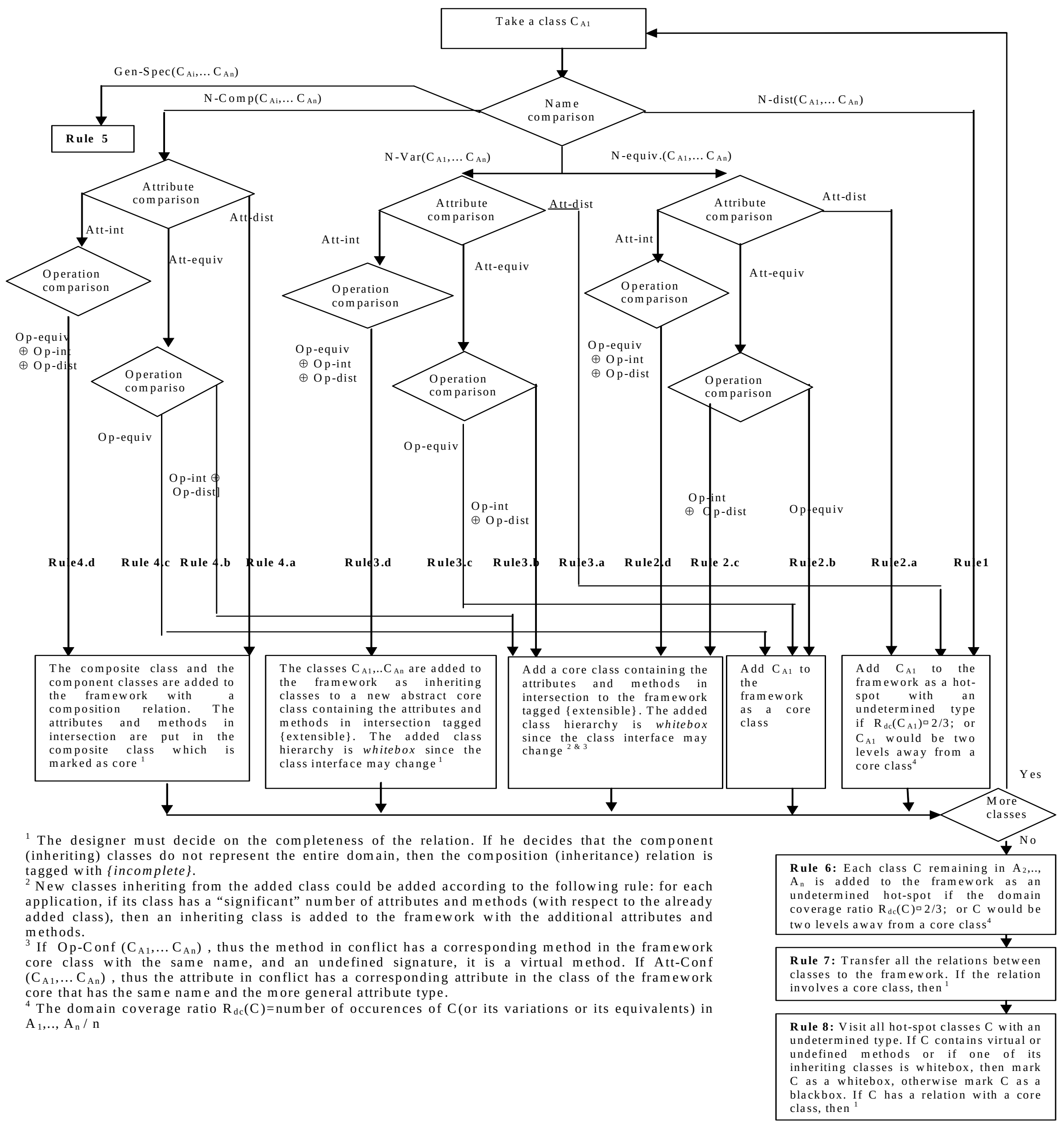
Further, *Rule 6* adds or ignores hot-spot classes according to the domain coverage ratio R_{dc} :

$$R_{dc}(C) = \frac{number\ of\ occurrences\ of\ C(or\ its\ \ \mathrm{var}iations\ or\ its\ equivalents)\ in\ A1,..,An}{number\ of\ applications}$$

Informally, this ratio is used to determine the reuse potential of a class. If a class is present in several applications, then it covers an important space of the framework domain; thus, it must be present in the framework hot-spot. On the other hand, if a class is present in few applications, it is too application specific; thus, if it is added to the framework, it may complicate unnecessarily the framework comprehension.

Sequence diagram unification

The design of the framework sequence diagram consists of five unification rules (shown in Figure 3) that rely on the semantic relations among classes adapted to objects and on name comparison of messages. The main idea of these rules is to unify sequence diagrams that correspond to the same or an equivalent scenario. That is, the process takes the “union” of all the sequence diagrams of the applications and marks any message as *{optional}* if it does not appear in all the applications.


Take a class C_{A1}
Gen-Spec(C_{Ai},… C_{An})
N-Comp(C_{Ai},… C_{An})
Name comparison
N-dist(C_{A1},… C_{An})
Rule 5
N-Var(C_{A1},… C_{An})
N-equiv.(C_{A1},… C_{An})
Attribute comparison
Att-int
Att-equiv
Att-dist
Operation comparison
Op-equiv ⊕ Op-int ⊕ Op-dist
Op-equiv
Op-int ⊕ Op-dist
Rule4.d
Rule 4.c
Rule 4.b
Rule 4.a
Rule3.d
Rule3.c
Rule3.b
Rule3.a
Rule2.d
Rule 2.c
Rule2.b
Rule2.a
Rule1
The composite class and the component classes are added to the framework with a composition relation. The attributes and methods in intersection are put in the composite class which is marked as core [1]
The classes C_{A1},..C_{An} are added to the framework as inheriting classes to a new abstract core class containing the attributes and methods in intersection tagged {extensible}. The added class hierarchy is *whitebox* since the class interface may change [1]
Add a core class containing the attributes and methods in intersection to the framework tagged {extensible}. The added class hierarchy is *whitebox* since the class interface may change [2 & 3]
Add C_{A1} to the framework as a core class
Add C_{A1} to the framework as a hot-spot with an undetermined type if $R_{dc}(C_{A1})$▫2/3; or C_{A1} would be two levels away from a core class[4]
Yes
More classes
No
Rule 6: Each class C remaining in A_2,.., A_n is added to the framework as an undetermined hot-spot if the domain coverage ratio $R_{dc}(C)$▫2/3; or C would be two levels away from a core class[4]
Rule 7: Transfer all the relations between classes to the framework. If the relation involves a core class, then [1]
Rule 8: Visit all hot-spot classes C with an undetermined type. If C contains virtual or undefined methods or if one of its inheriting classes is whitebox, then mark C as a whitebox, otherwise mark C as a blackbox. If C has a relation with a core class, then [1]

[1] The designer must decide on the completeness of the relation. If he decides that the component (inheriting) classes do not represent the entire domain, then the composition (inheritance) relation is tagged with *{incomplete}*.
[2] New classes inheriting from the added class could be added according to the following rule: for each application, if its class has a "significant" number of attributes and methods (with respect to the already added class), then an inheriting class is added to the framework with the additional attributes and methods.
[3] If Op-Conf (C_{A1},… C_{An}) , thus the method in conflict has a corresponding method in the framework core class with the same name, and an undefined signature, it is a virtual method. If Att-Conf (C_{A1},… C_{An}) , thus the attribute in conflict has a corresponding attribute in the class of the framework core that has the same name and the more general attribute type.
[4] The domain coverage ratio $R_{dc}(C)$=number of occurences of C(or its variations or its equivalents) in A_1,.., A_n / n

Fig. 2: Design of the framework class diagram

The design unification rules are implemented in the FUMLTool [Ayadi 2003]. This tool provides for the graphical representation of both applications and frameworks. Moreover, it manages the comparisons relations through a dictionary. In addition, it allows a semi-automatic generation of a framework design in F-UML based on the above presented unification rules. As the rules indicate the designer is probed only to identify the completeness of certain relations.

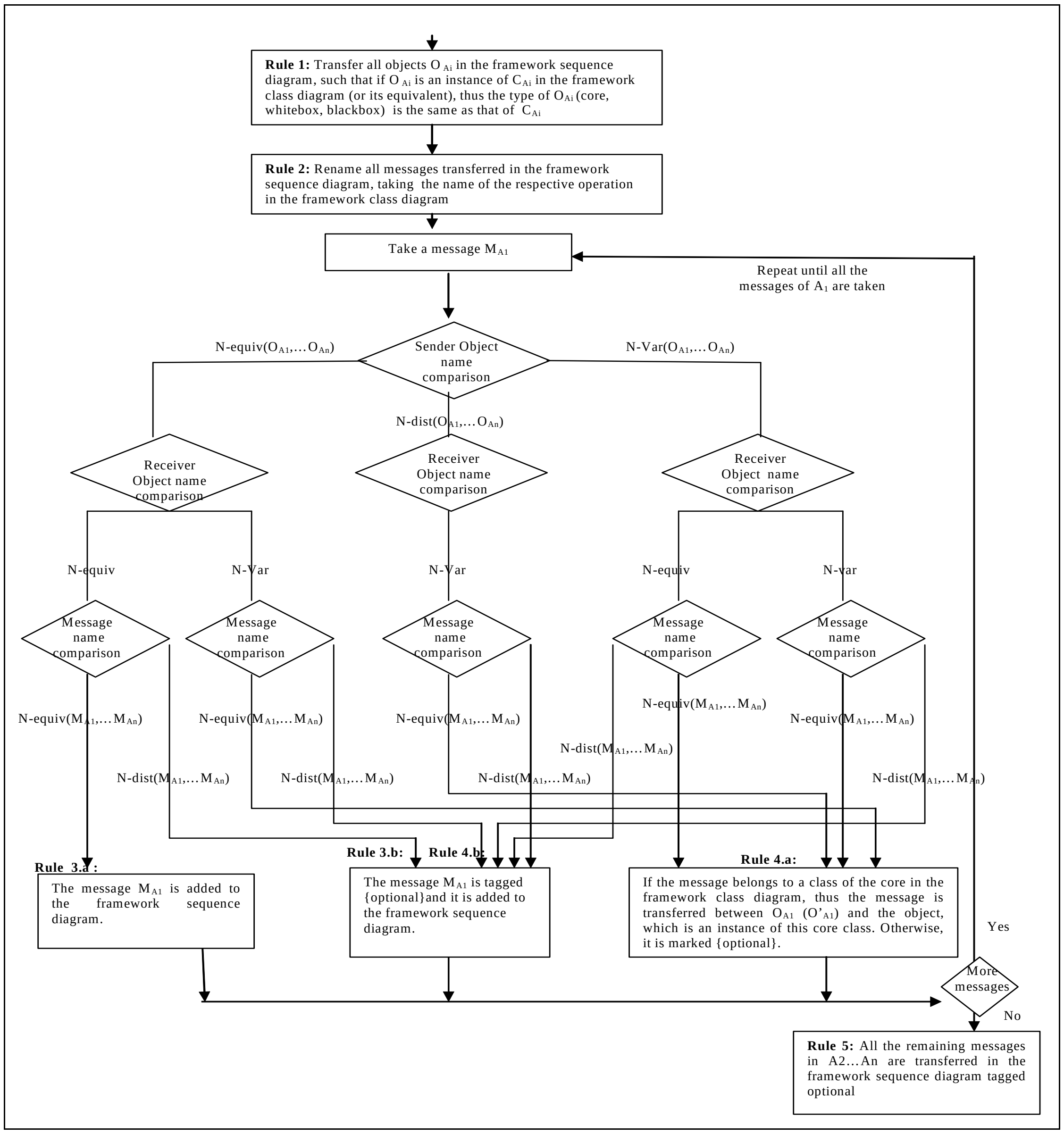

Rule 1: Transfer all objects O Ai in the framework sequence diagram, such that if O Ai is an instance of CAi in the framework class diagram (or its equivalent), thus the type of OAi (core, whitebox, blackbox) is the same as that of CAi
Rule 2: Rename all messages transferred in the framework sequence diagram, taking the name of the respective operation in the framework class diagram
Take a message MA1
Repeat until all the messages of A1 are taken
N-equiv(OA1,…OAn)
Sender Object name comparison
N-Var(OA1,…OAn)
N-dist(OA1,…OAn)
Receiver Object name comparison
Receiver Object name comparison
Receiver Object name comparison
N-equiv
N-Var
N-Var
N-equiv
N-var
Message name comparison
Message name comparison
Message name comparison
Message name comparison
Message name comparison
N-equiv(MA1,…MAn)
N-equiv(MA1,…MAn)
N-equiv(MA1,…MAn)
N-equiv(MA1,…MAn)
N-equiv(MA1,…MAn)
N-dist(MA1,…MAn)
N-dist(MA1,…MAn)
N-dist(MA1,…MAn)
N-dist(MA1,…MAn)
N-dist(MA1,…MAn)
Rule 3.a :
The message MA1 is added to the framework sequence diagram.
Rule 3.b:
Rule 4.b:
The message MA1 is tagged {optional}and it is added to the framework sequence diagram.
Rule 4.a:
If the message belongs to a class of the core in the framework class diagram, thus the message is transferred between OA1 (O'A1) and the object, which is an instance of this core class. Otherwise, it is marked {optional}.
Yes
More messages
No
Rule 5: All the remaining messages in A2…An are transferred in the framework sequence diagram tagged optional

Fig. 3: Design of the sequence diagram

5 APPLICATION: E-BROKER FRAMEWORK

To evaluate the F-UML language, process and tool, we designed an electronic commerce broker. The choice of electronic commerce domain has several motivations. One motivation is the expansion of e-commerce; a second motivation is the importance of the broker entity in this type of commerce; a third motivation is the software complexity of electronic brokers, which makes design reuse a judicious choice in order to benefit from the expertise gained in the design of particular e-broker applications.

In order to design the E-broker framework, we followed three steps. First, we took three e-broker applications designs: a book broker called Tunisia Book [Khelifi 2000], an auto broker called Auto Broker [Birkes 1995] and a broker for antiquities called Antique Broker [Antique 2002]. Secondly, we constructed the relation dictionary used in the comparison rules of the design process. Finally, we used the framework generation function of the F-UMLTool to obtain the E-broker framework. Due to space limitations, we next present a simplified version of this case study.

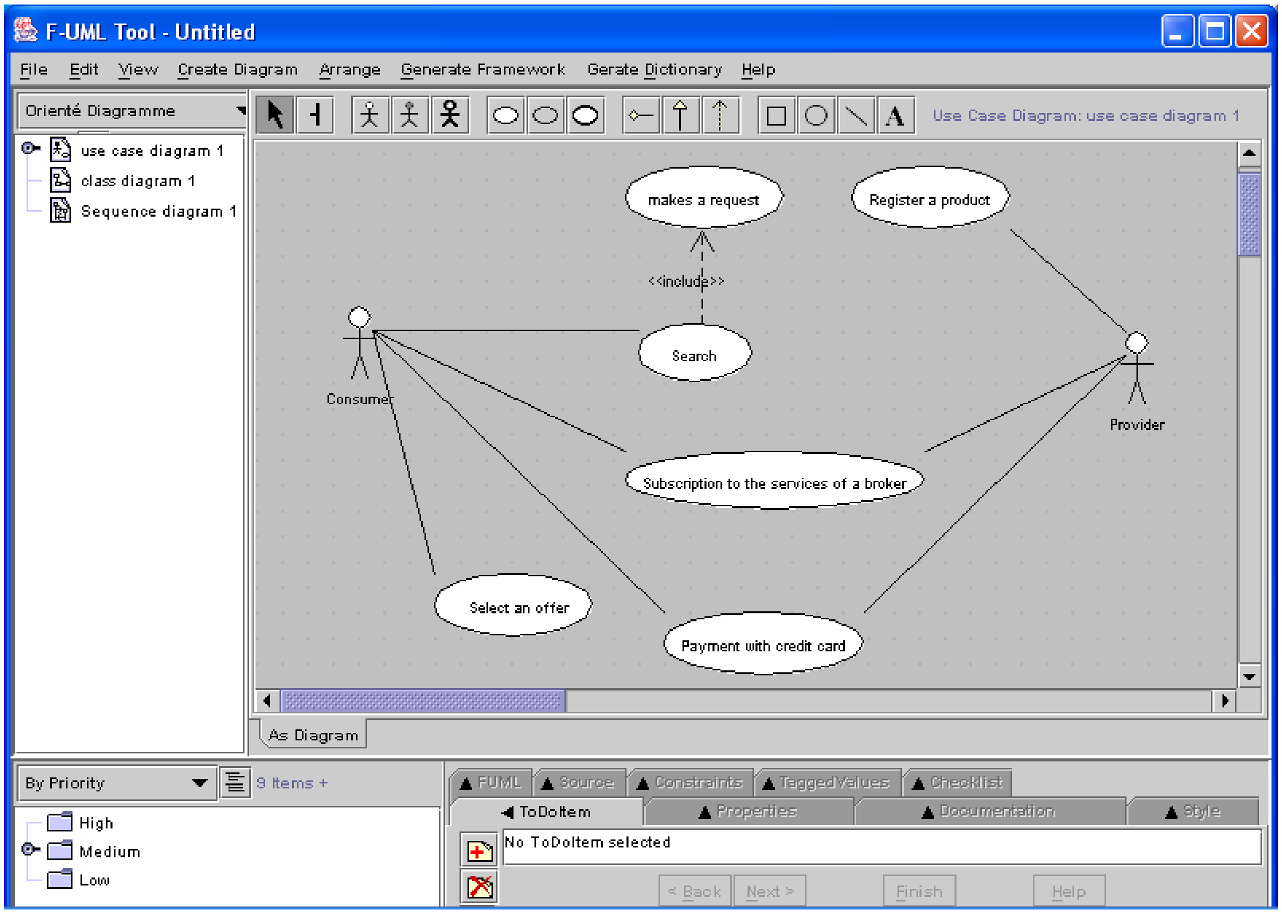


Fig. 4: The use case diagram of Tunisia Book

Use case generation

Figure 4 presents part of the use case diagram of Tunisia Book. As illustrated in this figure, the actors that use the brokerage services are the Consumer and the Provider. One of the functionalities assumed by this application is the consumer/provider subscription to the broker. Another functionality is the Provider registration of its products for sale and the Consumer product request submission. The functionality of the broker is to search for the best offer and to return the list of books found. The consumer evaluates the list of found offers and decides which books to buy and pays for them.

Figure 5 presents part of the use case diagram of the Auto broker. As illustrated in this figure, the actors that use the system are Buyer and Seller, two variants of the actors in Tunisia Book. Unlike Tunisia Book, this broker does not require a subscription. Figure 6 presents part of the use case diagram of the Antique Broker.

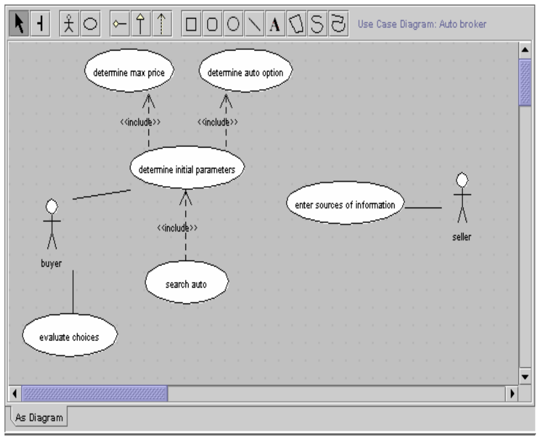


Fig.5: The use case diagram of Auto broker

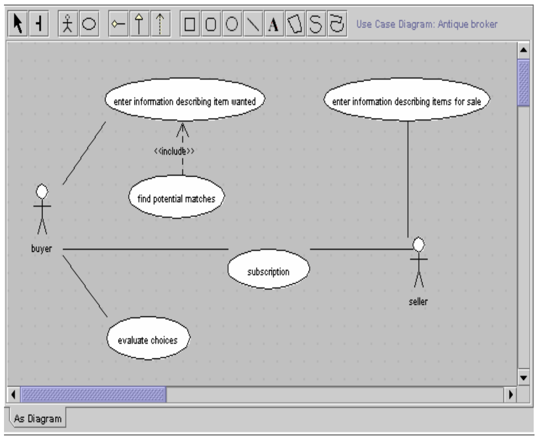

Use Case Diagram: Antique broker
enter information describing item wanted
enter information describing items for sale
<<include>>
find potential matches
buyer
subscription
seller
evaluate choices
As Diagram

Fig. 6:. The use case diagram of Antique Broker

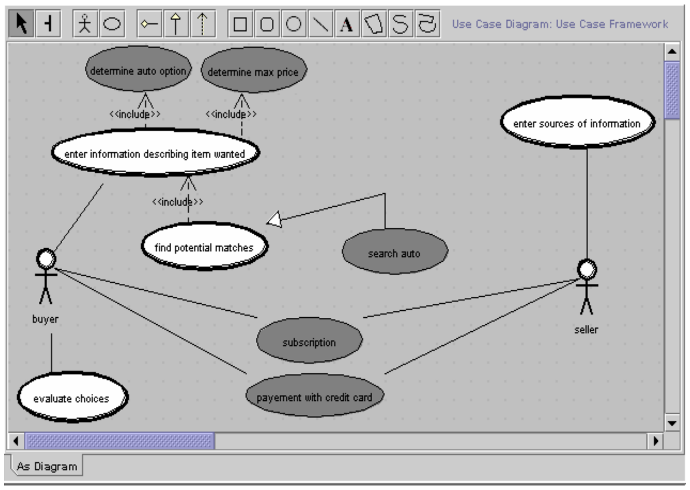


Fig. 7: The framework use case diagram

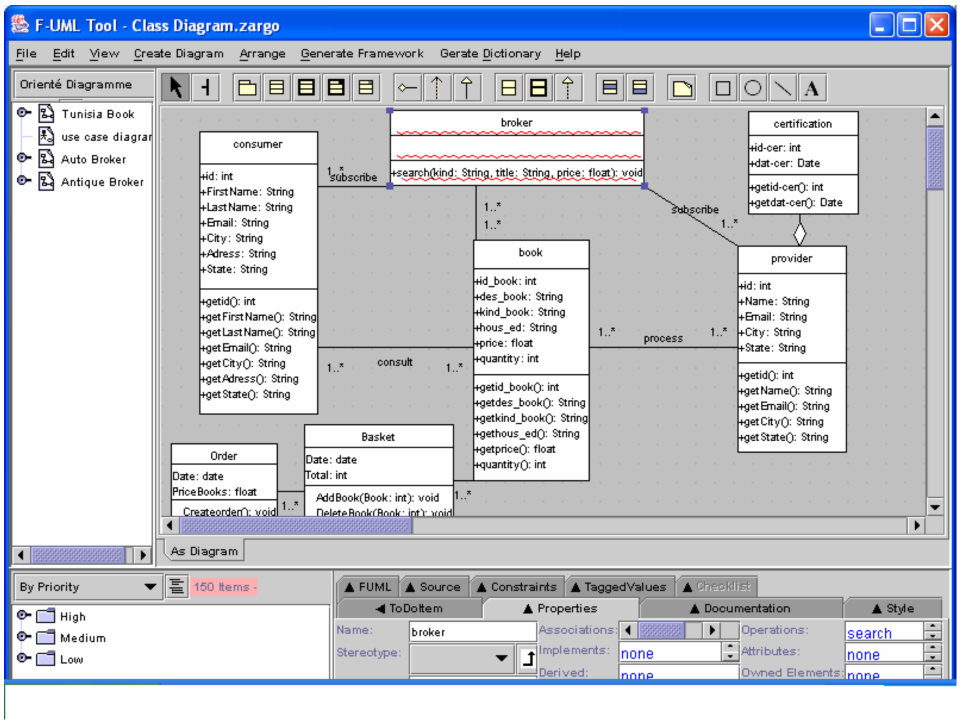
The generated use case diagram of the E-broker framework is shown in Figure 7. It has been obtained by applying the unification rules as indicated in Table 4.

Rule		Framework
Criteria	N°	
N-equiv(*Provider*$_{A1}$, *Seller*$_{A2}$ *Seller*$_{A3}$)	1	The actor *Seller* is added to the framework core
N-equiv(*Consumer*$_{A1}$, *Buyer*$_{A2}$ *Buyer*$_{A3}$)	1	The actor *Buyer* is added to the framework core
N-equiv(*makereqest*$_{A1}$,*DetermineInitialParameter*$_{A2}$ *Enter Information Describing Item wanted* $_{A3}$)	1	The use case *Enter Information Describing Item wanted* is added to the framework core
N-equiv(*EntersourceInformation* $_{A1}$,*Register product*$_{A2,}$ *Enter Information Describing Item for sale* $_{A3}$)	1	The use case *Enter sources of information* is added to the framework core
N-equiv(*Evaluate choices* $_{A1}$ *select an offer* $_{A2}$ *select an offer* $_{A3}$)	1	The use case *Evaluate choices* is added to the core
N-equiv(Search $_{A1}$*Find potential match* $_{A2}$*Search*$_{A3}$*)*	1	The use case *Search* is added to the core
Gen-Spec(Search $_{A1}$, *Search auto* $_{A2}$ *Search* $_{A3}$)	3	A generalization between the use case *Find potential matches* and *search auto* is added
N-dist(subscription to the services of a broker) *N-dist(Subscription)* *N-dist (Determine max price)* *N-dist (Payment with credit card)*	4	The use cases *subscription, determine max price* and *Payment with credit card* are added as framework hot-spots

Table 4: Design of the use case diagram of the framework broker

Class diagram generation

Figure 8 presents part of the class diagram of Tunisia Book. The Broker manages the Consumer and Provider subscriptions. Furthermore, the association between the Consumer class and the Book class represents the consumers' evaluation and selection of found offers.



consumer: +id: int, +FirstName: String, +LastName: String, +Email: String, +City: String, +Adress: String, +State: String; +getid(): int, +getFirstName(): String, +getLastName(): String, +getEmail(): String, +getCity(): String, +getAdress(): String, +getState(): String

broker: +search(kind: String, title: String, price: float): void

certification: +id-cer: int, +dat-cer: Date; +getid-cer(): int, +getdat-cer(): Date

book: +id_book: int, +des_book: String, +kind_book: String, +hous_ed: String, +price: float, +quantity: int; +getid_book(): int, +getdes_book(): String, +getkind_book(): String, +gethous_ed(): String, +getprice(): float, +quantity(): int

provider: +id: int, +Name: String, +Email: String, +City: String, +State: String; +getid(): int, +getName(): String, +getEmail(): String, +getCity(): String, +getState(): String

Order: Date: date, PriceBooks: float; Createorder(): void

Basket: Date: date, Total: int; AddBook(Book: int): void, DeleteBook(Book: int): void

Name: broker
Stereotype:
Associations:
Implements: none
Derived: none
Operations: search
Attributes: none
Owned Elements: none

Fig. 8: The class diagram of Tunisia Book

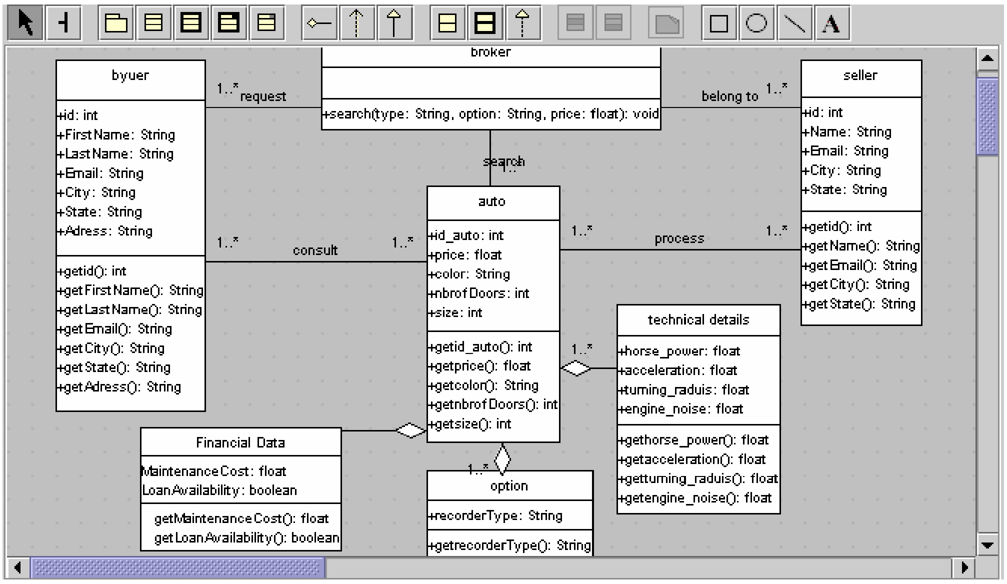


Fig. 9: The class diagram of Auto broker

Figure 9 partially presents the class diagram of Auto Broker. Similar to the previous broker class diagram, the class diagram of Auto broker has the classes Broker, Seller and Buyer. In addition, it has the class Auto and its aggregate classes: Options and Technical Details.

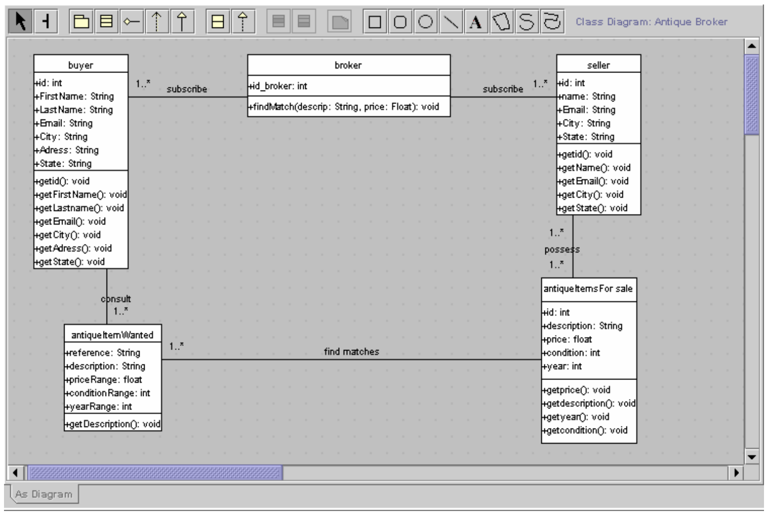


Fig. 10: The class diagram of Antique Broker

Rule	N°	Framework
N-equiv(Broker$_{A1}$, Broker$_{A2}$ Broker$_{A3}$)ƒ Att-equiv(Broker$_{A1}$, Broker$_{A2}$ Broker$_{A3}$) ƒ Op-dist(Broker$_{A1}$, Broker$_{A2}$, Broker$_{A3}$)) ƒ Op-conf(Broker$_{A1}$, Broker$_{A2}$, Broker$_{A3}$)	2.d	*Broker* is added to the framework core with the tag {extensible}. The method *Search* is an operation in conflict, thus it has an undefined signature. Broker is a whitebox hot-spot.
N-equiv(Provider$_{A1}$, Seller$_{A2}$ Seller$_{A3}$)ƒ Att-equiv(Provider$_{A1}$, Seller$_{A2}$ Seller$_{A3}$) ƒ Op-equiv(Provider$_{A1}$, Seller$_{A2}$ Seller$_{A3}$)	2.b	*Seller* is added to the framework core
N-equiv(Consumer$_{A1}$, Buyer$_{A2}$ Buyer$_{A3}$)ƒ Att-equiv(Consumer$_{A1}$, Buyer$_{A2}$ Buyer$_{A3}$)ƒ Op-equiv(Consumer$_{A1}$, Buyer$_{A2}$ Buyer$_{A3}$)	2.b	*Buyer* is added to the framework core
N-var(Book $_{A1}$, AntiqueItemsForsale$_{A2}$ Auto$_{A3}$)ƒAtt-int (Book$_{A1}$, AntiqueItemsForsale$_{A2}$ Auto$_{A3}$)ƒ Op-int var(Book $_{A1}$, AntiqueItemsForsale$_{A2}$ Auto$_{A3}$)	3.d	*Auto, Book, AntiqueItemsForsale* are added to the framework as inheriting classes to the class *Product*. The class *Product* has the common attributes and methods and is tagged with {extensible}. This hierarchy constitutes a whitebox hot-spot in the Framework. We added the tag {incomplete} to the hierarchy because the application does not contain all possible variants of Product.
N-dist(Certification $_{A1}$)	1	The class *Certification* is a hot-spot
N-dist(Basket $_{A1}$)	1	The class *Basket* is a hot-spot
N-dist(Order $_{A1}$)	1	The class *Order* is a hot-spot
	6	The remaining classes: Option, Technical Details, Antique Item wanted are transferred in the framework as blackbox hot-spots
	7	The relations between classes are transferred in the framework

Table 5: Design of the class diagram of the framework broker

As illustrated in Figure 10, which presents the class diagram of Antique Broker, the common classes with the previous applications are Broker, Buyer and Seller. The main difference is the class AntiqueItemWanted which is a hot-spot.

The framework class diagram (Figure 11) has been obtained by applying the unification rules shown in Table 5. The application A1 is Tunisia Book since it contains the minimum number of classes, A2 is Antique Broker and A3 is Auto Broker. During the generation of the class diagram, F-UMLTool produced the diagram shown in Figure 11 with the inheritance relation between *auto*, *book*, *antiqueitemwanted* and *Product* as undecided (Rule 3.d). We have decided that it is incomplete since the inheriting classes does not cover all the domain.

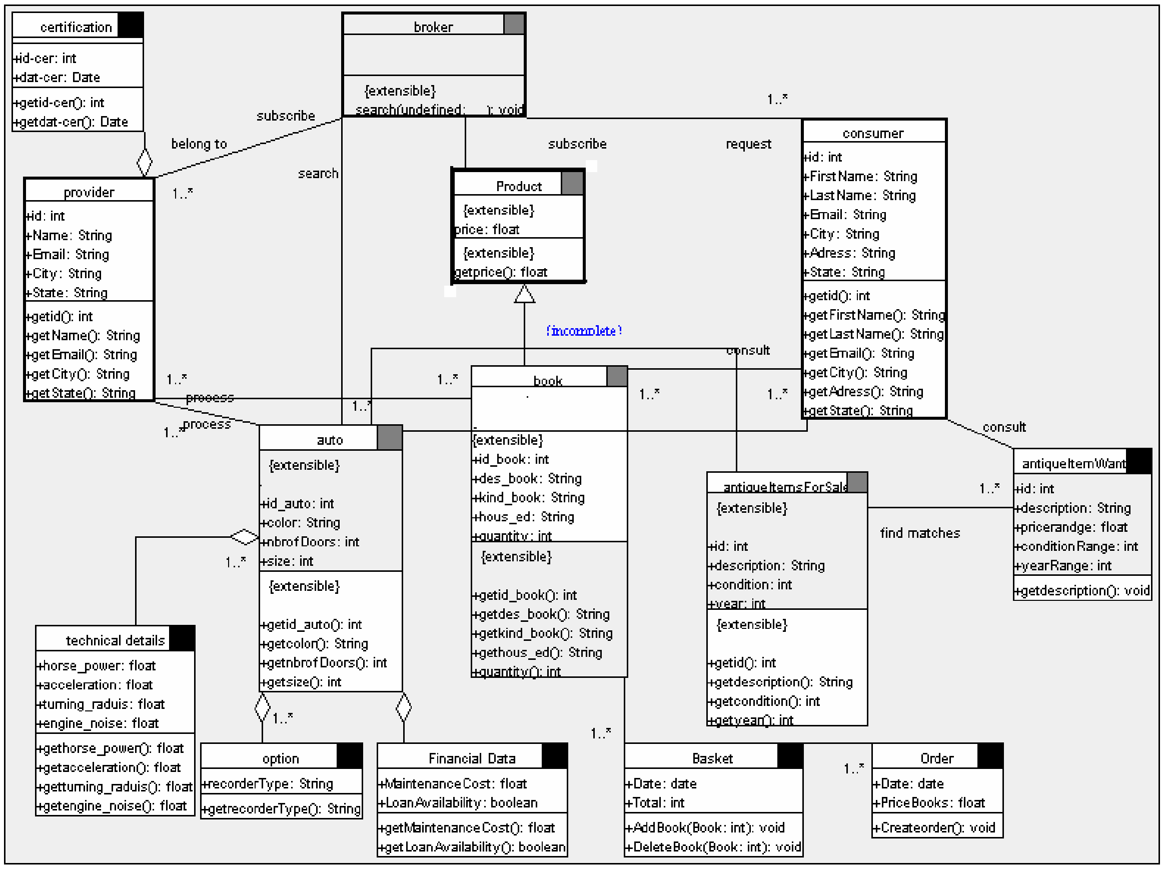


Fig. 11: The framework class diagram

Sequence diagram generation

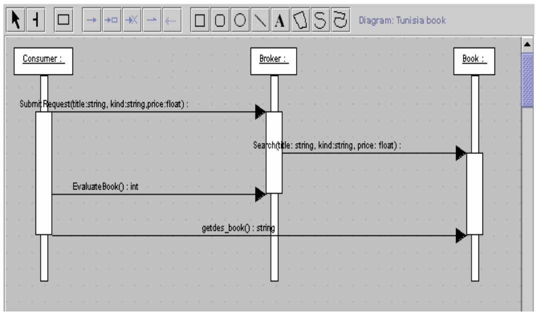


Diagram: Tunisia book
Consumer :
Broker :
Book :
SubmitRequest(title:string, kind:string,price:float) :
Search(title: string, kind:string, price: float) :
EvaluateBook() : int
getdes_book() : string

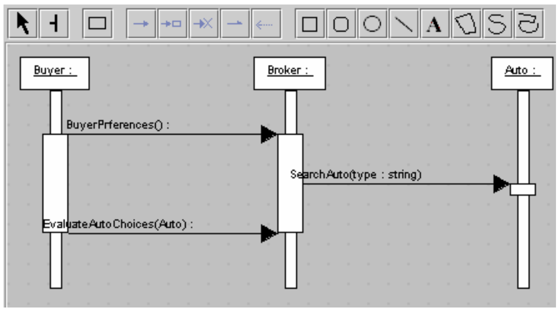


Fig. 12: A sequence diagram of Tunisia book

Fig. 13: A sequence diagram of auto Broker

Figure 12 presents the simplified sequence diagram of Tunisia Book, which corresponds to the scenario “Search and buy”. Figure 13 presents a part of a sequence diagram Auto Broker, which corresponds to the scenario “Search and evaluate auto”. Figure 14 presents a part of the sequence diagram of the antique Broker corresponding to the scenario “Search and evaluation”.

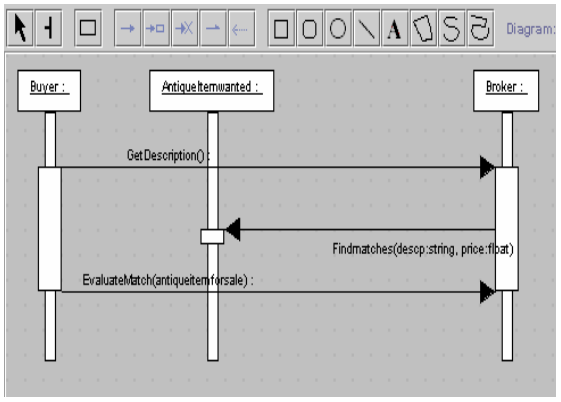


Diagram:
Buyer :
AntiqueItemwanted :
Broker :
GetDescription() :
Findmatches(descp:string, price:float)
EvaluateMatch(antiqueitemforsale) :

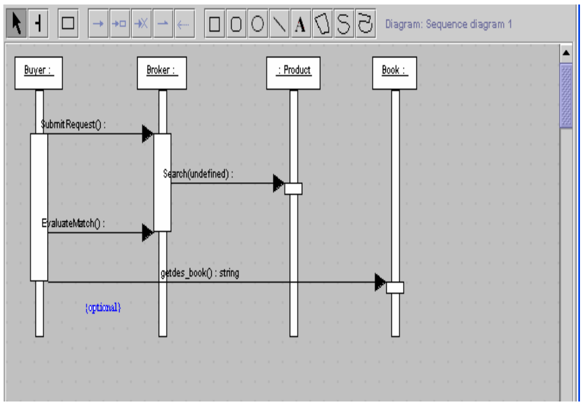


Fig. 14: A sequence diagram of antique Broker

Fig. 15: The framework sequence diagram

The Framework sequence diagram corresponding to the scenario "Search and evaluation" (Figure 15) was automatically derived using the semantic comparison between the message' names and types and between the sending and receiving object names.

6 EVALUATION OF THE FBDM METHOD

In addition to the E-commerce domain, we have evaluated the FBDM method and its associated toolset in the graphical drawing editor domain [Bouassida 2005]. This domain was chosen since a popular and mature framework already exists: the JHotDraw framework (about one hundred classes) [Gamma 1997]. Furthermore, JHotDraw was chosen in order to have a comparison basis for the generated framework since several applications were derived from it.

On one hand, both case studies showed that the F-UML notation facilitates the distinction between the core of the framework, which must be present in any application derived from it, and its variable parts (hot-spots). This in turn can guide the designer in estimating the degree of reuse the framework can offer. On the other hand, both case studies showed that the FBDM design process generates a framework that contains the whole framework core (e.g., Buyer, Seller, Product for E-Broker). However, the design process produces a framework with (possibly too many) application specific increments i.e., classes specific to a particular application (e.g., Antique item wanted, Option, Technical details in E-Broker). These latter could complicate the comprehension of the framework and hence impede its reuse. However, the F-UML notation helps by visually distinguishing these details. Thus, when reusing a framework, the designer can first focus on the core, and later he/she can choose to examine or ignore the hot-spots. Moreover, some of the framework internal increments (i.e., classes belonging to the library accompanying the framework) are not deduced by our design process since they are absent in the original applications. However, the process puts the tag {incomplete} wherever the designer can add details (classes, attributes, or methods). While this may reduce the number of classes reused (as a reuse measurement), the designer is at least advised of the places he/she is expected to focus his/her design effort.

Overall, we can note that the degree of details produced in the generated framework depends on the level of domain coverage in the unified applications. Finally, the F-UML Tool was vital in the design process, especially in managing the complexity of applying the rules face to the size of the diagrams, in particular for the JHotDraw case study.

7 CONCLUSION

This paper first reviewed current framework design languages and processes. Secondly, it presented a framework design method that offers a UML-based design language and a systematic bottom-up design process. The presented FBDM method is distinguished from existing methods in three ways: 1) its UML-profile language visually distinguishes between the framework core, whitebox and blackbox hot-spots; 2) its design process is well-defined through a precise set of unification rules that identifies automatically the core and hot-spots; and 3) its toolset provides for the graphical representation of applications and the semi-automatic generation of a framework. The FBDM design method was illustrated through the design of a framework for brokers of e-commerce. It was evaluated through two case studies in the e-commerce and graphical editor domains.

We are currently investigating how to generate automatically the dictionary of the semantic relations in the design process. Future works include the development of a module for the generation of the pattern diagram. The module would propose the patterns adapted to a design and the framework designer would decide which pattern fits the problem.

REFERENCES

[Antique02] http://www.Antiqnet.com

[ArgoUML02] ArgoUML, http://arguml.tigris.org

[Aksit99] M. Aksit, B. Tekinerdogan, F. Marcelloni and L. Bergmans: Deriving Object-Oriented Frameworks from Domain Knowledge, Building Application Frameworks: Object-Oriented Foundations of Framework Design, M. Fayad, D. Schmidt, R. Johnson (Eds.), John Wiley & Sons Inc., pp. 169-198, 1999.

[AYADI03] T. Ayadi, N. Bouassida, H. Ben-Abdallah, F. Gargouri, D. Slimane: F-UMLTool: A Tool for Object-Oriented Framework Design, Int'l conference on Industrial Engineering and Production Management (IEPM'03), Portugal, 26-28 May 2003.

[Birkes95] A. Y. Birkes, C. H. Hsiung, T. Cohen and R. E. Fulton: A Prototype Multimedia Auto Broker, ASME Computers in Engineering Conference, Proceedings of the engineering Database Symposium Boston, MA, Sept 17-21, 1995.

[Bosch97] J. Bosch, P. Molin, M. Mattsson, P. O. Bengtsson: Object oriented frameworks: problems & experiences, http://www.ide.hk-r.se/~michaelm/papers/ex-frame.ps , 1997.

[Bouassida01] N. Bouassida, H. Ben-Abdallah, and F. Gargouri: A UML based Design Language for Framework Reuse, 7th International Conference on Object-Oriented Information Systems (OOIS'01), Calgary, Canada, 2001.

[Bouassida02] N. Bouassida, H. Ben-Abdallah, and F. Gargouri, A. Ben-Hamadou: A stepwise Framework Design Process, IEEE International Conference on Systems Man and Cybernetics, 07-09 October, Hammamet, Tunisia, 2002.

[Bouassida05] N. Bouassida, H. Ben-Abdallah, F. Gargouri, A. Ben-Hamadou: "Evaluation of a framework design method", submitted to *Information Sciences for Decision Making Journal* (ISDM), 2005.

[Bouassida03b] N. Bouassida, H. Ben-Abdallah, and F. Gargouri, A. Ben-Hamadou: F-UML: a design language for frameworks and its formal specification, International conference on Software Engineering and Formal Methods (SEFM'2003), Australia, Brisbane, 26-29 September, 2003.

[Fontoura00a] M.F. Fontoura, S. Crespo, C.J. Lucena, P. Alencar, D. Cowan: Using viewpoints to derive Object-Oriented Frameworks: A case study in the web education domain, Journal of Systems and Software (JSS) Elsevier Science, 54 (3), 2000.

[Fontoura00b] M.F. Fontoura, W. Pree, B. Rumpe: UML-F: A Modeling Language for Object-Oriented Frameworks, Proceedings of European Conference on Object Oriented Programming (ECOOP 2000), Springer-Verlag, 2000.

[Gamma95] E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design patterns: Elements of reusable Object Oriented Software, Addisson-Wesley, Reading, MA,1995.

[Gamma97] E. Gamma, T. Eggenschwiler, http://www.jhotdraw.org

[Johnson88] R. E. Johnson, B. Foote: Designing reusable classes, Journal of Object Oriented Programming, Vol 1, N°2, June/July 1988,pp 22-35.

[Khelifi00] A. Khelifi, Les Brokers pour le commerce électronique : analyse de l'existant, développement d'un prototype, rapport de fin d'études, Faculté des sciences de Tunis, 2000.

[Koskimies95] K. Koskimies, H. Mossenback: Designing a framework by stepwise generalization, 5th European Software Engineering Conference, Barcelona. Lecture Notes in Computer Science 989, Springer-Verlag, 1995.

[OCL97] Object Constraint Language, version 1.1, September, 1997.

[Pree94] W. Pree: Meta-patterns: a means for capturing the essentials of object-oriented designs, Proceedings of the 8th European Conference on Object Oriented Programming, Bologna, Italy, 1994.

[Rational01] Rational Software, URL: http://www.rational.com/UML/

[Reenskaug96] T. Reenskaug : Working with objects, Greenwich : Manning, 1996.

[Riehle00] D. Riehle, Framework design: A Role modelling approach, Dissertation N° 13509, ETH, Zurich, 2000. http://www.riehle.org/diss/

[Roberts96] D. Roberts, R. Johnson, Evolving Frameworks: A pattern language for Developing Object Oriented Frameworks, Proceedings of the third conference on pattern languages and programming, Montecilio, Illinois, 1996.

[Rumbaugh91] Rumbaugh et al., Object Oriented Modelling and design, Prentice Hall, 1991.

[Sanada02] Y. Sanada, R. Adams: Representing Design Patterns and Frameworks in UML-Towards a Comprehensive Approach, Journal of Object Technology, Vol. 1, N°2, July-August 2002.

[Schmid97] H. A. Schmid: Systematic framework design by generalization, Communications of the ACM, Special issue on Object Oriented Application frameworks, vol. 40, no. 10, October 1997.

About the authors

Hanene Ben-Abdallah received a Ph.d. in Computer and Information Science from theUniversity of Pennsylvania, Philadelphia, PA. She is currently Assistant Professor at the Department of Computer Science of Faculté des Sciences Economique et de Gestion de Sfax at the University of Sfax, Tunisia. Her research interests include formal method application and architecture reuse techniques. Her e-mail address is hanene@gnet.tn.

Nadia Bouassida is preparing a Doctorate degree in Computer Science at the Faculté des Sciences de Tunis, Tunisia. She is a Teaching Assistant at the Institut Supérieur d'Informatique et du Multimédia of The University of Sfax, Tunisia. Her e-mail address is nadia.bouassida@isimsf.rnu.tn.

Faiez Gargouri is an Associate Professor at the Institut Supérieur d'Informatique et du Multimédia of The University of Sfax, Tunisia. He worked in Paris 5 and Paris 13 universities. His research interests include object-oriented software design, Data Warehouses and conceptual reuse. His e-mail address is faiez.gargouri@fsegs.rnu.tn.

Abdelmajid Ben-Hamadou is Professor of Computer Science and the director of the Institut Supérieur d'Informatique et du Multimédia of The University of Sfax, Tunisia. He has been the director of the LARIS reserach laboratory since 1983. His research interests include automatic processing of Natural Language, object-oriented design and component software specification. His e-mail address is abdelmajid.benhamadou@isimsf.rnu.tn.