Vrije Universiteit Brussel - Belgium
Faculty of Sciences
In Collaboration with Ecole des Mines de Nantes - France
2001

ARCom (Another Reusable Component Model)

A Thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
(Thesis research conducted in the EMOOSE exchange)

By: Gustavo J. Bobeff

Promotor: Prof. Theo D'Hondt (Vrije Universiteit Brussel - Belgium)
Co-Promotor: Mourad Oussalah (University of Nantes - IRIN - France)
Co-Promotor: Annya Romanczuk-Réquilé (Ecole des Mines de Nantes - France)

I would like to thank and dedicate this work

to **Carla** and **Santiago** because they are my spiritual support to do these things, and the most important beings in my life.

to my father, **Cristo**, because from him I inherit a progressive behavior.

to my family and friends for be there, specially to **Andres Farias** because he has known to help me when I needed it, and also to **Sinagi**.

to my advisors, **Annya Romanczuk** and **Mourad Oussalah**, because they have been patients with me and good guiders, thanks.

to **Gustavo Rossi** because he put his trust in me.

to all the new friends who made pleasant our stay, **Sofie Goredis**, **Patricio Salinas Caro**, **Victor Hugo Arroyo**, **Kristof De Vos**, **Marc Ségura-Devillecaise**, **David Würth** and **Liang Peng**.

Abstract

Component-based software development has proven effectiveness for systems implementation in well-understood application domains, but is still insufficient for the creation of reusable and changeable software architectures. Applications build on components that prevent the dilution of design decisions seems to be a suitable solution. The aim of this thesis is the definition of a reusable component model where the design decisions can be kept through any composition mechanism. The component design used for our model, named ARCoM(Another Reusable Component Model) was conceived for software design, but always keeping a mapping with the implementation artifacts in object-oriented programming, that is, classes and relationships.

Key words: Component-based Software Development, Software Reuse, Software Composition, Object-oriented Design, Reuse Design.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Constructing software by glutting code from different sources, and coding from scratch others, has been a technique highly used since much time ago. For instance, an application could be built by coping functions or modules already implemented in other systems. But, software development from applying this method does not guarantee a system's life for a lengthy period of time, since some software quality factors can be damaging (e.g.maintainability). Due to reusing software blocks became an usual strategy in system development, systemic and formal model appeared in order to provide a solid framework for *software reusing*.

A reused block has been named in many ways, and generally the label depends on the environment where the reusing takes place. For instance, a UNIX's user thinks in terms of reusing when he executes a piped command, that is, a chain of commands where every element into it takes the input form the precedent one and generates output for the antecedent element. In this example, the set of instructions organized in the command represents the reused block. But the current name comes form the electronic circuite industry, where the electronic devices are built by assembling *components* from which is known their requirements to work and their utility provided.

Component design implies decisions like, for example, the degree of allowance available for the user to adapt its internal structure, or reuse as it is. Those design alternatives are know as white-box and black-box components respectively. The successful models have shared features respecting the structures used to represent the components and goals for standardizing to a unique model, as well. The nature of the component's idea satisfies, in many aspects, the object oriented paradigm, that is, encapsulation of common functionality, behavioral declaration (interface definition) and representation techniques. Hence many existent component models are object-oriented since they based its representation on the artifacts defined into the paradigm. Nowadays it is possible to access to this emerging technology just by using a *component models*, namely, *Component Object Model (COM)*, *JavaBeans* or *CORBA*. The heterogenous models evidence that a maturing process must happen for getting a slightly unified understanding as it took place in object-oriented paradigm.

So far it seems to be a nice story because to undertake a component-based software development is possible by assembly prefabricated building block, and aside from other capabilities, they could be adapted to fit in a new situation, keeping a certain degree of independency from its original developing environment [KS98]. But the some differences raise when we try to get a definition for *component composition*. To avoid any conflict, component composition can be defined as the process by means reusable components are put together to solve a particular problem. Chapter 2 covers elements that should be present in any discission on component-based development.

To clarify the scenario where software reusing is taken into account, it turns out suitable to distinguish two roles apart from those already considered for the classical software development, namely, the producer and the consumer. They are the roles that deal with the reusable component development and component-based development respectively. Chapter 3 describes the engineering approaches that we apply for drawing the model suggested.

When an analysis is required, and we use, for example, a top-down approach, we start from a high abstraction level for describing the current problem and begin to decompose the current problem based on outstanding aspects. During the analysis the focus turns on searching solutions for these subproblems, that may be solved by using components. But the main problem in component-based software (the same is endured by other systems independently from the development technique used), is the disability of preserving information about the resulting software architecture. Successful software architectures usually arise from a continuous reassessment of design alternatives and redistribution of responsibilities among system components. To achieve this in a component-based environment, deep insight into the components' design is required [KS98].

Components as part of a puzzle should provide not only information about their structure and behavior but also makes a contribution to the fulfilling and meaning of the resulting component. It should be useful to know which set of these pieces form a particular image, which of them belong to more than one identifiable image, etc. We change from implementation solution, software component level, to design solution providing, *design component* level [KS98].

The aim of this thesis is the definition of a reusable component model where the design decisions can be kept through any composition mechanism. The component design used for our model, named ARCoM(Another Reusable Component Model) was conceived for software design, but always keeping a mapping with the implementation artifacts in object-oriented programming, that is, classes and relationships. Chapter 4 depicts the fundamentals of ARCoM components applying the conceptual framework introduced in Chapter 3.

In order to provide a technical support for the model defined in this work, we have designed a tool family that makes possible to manipulate ARCoM components. A brief description about the responsibilities of family's members is introduced in Chapter 5.

Finally, implementation issues related to the prototype's implementation, the file formats used to store component's descriptions, and an example in details, are introduced in Appendix A, B and C respectively.

Chapter 2

Wearing Concrete Shoes

To define a robust starting point of our discussion we analyze the central concepts, *software reusing* and *software components*. In this section we collect many definitions for both ideas and break down them to extract their main intentions.

2.1 Software Reuse

2.1.1 Definitions

The following list enumerates several definitions *software reuse*:

- For Freeman [Fre83] reuse is the use of any information which a developer may need in the software creation process.
- For Tracz [Tra95] reuse is the use of software that was designed for reuse.
- For Braun [CJ94] reuse is the use of existing software *components* in a new context, either elsewhere in the same system or in another system.
- For Krueger [Kru92] software reuse is the process of creating software systems from existing software rather than building them from scratch.
- For Peterson [Pet91] reuse can be defined as application of existing solution to the problems of system development, and reuse stands

for reuse-as-is and reuse-with-modify which are sometimes called utilization or exploitation.

- Jacobson [JGJ97] states that reuse is the further use or repeated use of an artefact, where an artifact is designed for use outside of its original context to create new systems.
- Johnson [Joh91] defines software reuse as the process of incorporating into a new product any of the following: previously tested code, previously develop requirements specification, or previously tested plans, data, and procedures.
- In [ABvdSB94] Aksit said that the application code must be reused separately from its real-time specifications. This promotes the reuse of both, application code and real-time specification.

The aforementioned definitions focuss on several aspects, that is, *assets*, *approaches*, *intentions* and *benefits*. Although they come from different origins, they share many attribute. The list below describe what *software reuse* is, by using the aspects named previously:

Assets : One thing considered to be reuse can be either an abstract asset (e.g. information, solution description, plans, etc.) or a concrete asset (e.g. existing software, code, procedures). From many points of view, all of these terms agree with the *component* concept (see section 2.2), even when the abstraction degree is significative.

Approach : The reuse of life cycle objects, primarily code, is often done in an informal and haphazard way. Thus reuse process can be faced using mainly two strategies [Szy98], the first one is called *opportunistic reuse* which does not imply a specific software engineering process, and the second one is called *systemic reuse*, which includes specific software engineering process that we will explain in section 3.

Intentions : Some synonyms were used to express the meaning of *software reusing* [Hem93], commonality (reusability of languages for many people), portability (reusability of a program of software tool on many computers), modularity (reusability of software components in large applications), maintainability (reusability of the unchanged part of a program when a small change has been made), evolution (reusability of a system as it evolves in response to changing needs).

Benefits : Software reuse has a positive impact on software quality (error fixes accumulates from reuse to reuse), as well as on software cost (maintenance cost, time to market, training cost, etc, have to saved) , and productivity (less code has to be written).

Software reuse proposes benefits in development phases but to achieve or to maximize them is needed a systemic approach, however. A careful specification of what a component is, also guaranties take advantage from this activity. In the next section we debate about component definitions.

2.2 Software Components

2.2.1 Definitions

The vagueness to define the component definitions have been taken as the criteria to order the above list:

- From Booch[Boo87] a reusable software component is a logically cohesive, loosely coupled module that denotes a single abstraction.
- In [JCJO92] Jacobson et al, components are already implemented units that we use to enhance the programming languages constructs. These are used during programming and correspond to the components in the building industry.
- Reusable software components, in[Gro97], are self-contained, clearly identifiable pieces that describe and/or perform specific functions, have clear interfaces, appropriated documentations, and a defined reuse status.
- Meyer[Mey99] defines widely a software component as a program element with the following two properties:i) it may be used by other program elements, or clients, and ii) the clients and their authors do not need to be known to the component's authors.
- Szyperski in [Szy98] defines software component as a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.
- Software components are defined as prefabricated, pretested, self-contained, reusable software modules - bundles of data and procedures - that perform specific functions, for Sametinger in [Sam97].

2.2.2 Properties

By filtering the previous definitions may be thought over a set of generic properties to describe a component, namely, *modularity*, *reuse abstraction* and *independence.*

Modularity : A modular aspect can be considered since a component is seen as a unit that performs a defined function, in many cases, depending on the reusing abstraction (see Reuse Abstraction below). Encapsulation does not seem a suitable term because it reveals a quite restrictive viewpoint on the accessibility to the internal structure of the components, and not always that is true. Another way for the component specification can be through the use of *interfaces.* In this scenario a client of a software component hangs on the component specification, but not on its implementation. The use of interface specification is the common mechanism to define the component's bound. Components include data which represent their states and functions for processing these data, feature that fulfills an object definition in object technology.

Reuse Abstraction : A software component is associated to an abstraction which rules the accessibility degree to the internals when the component is reused.

- **Black-box**: The client can not see beyond the interface. In this cases reusing has to relay only in the interface specification, for example, application programming interfaces (APIs). Part of the implementation is revealed as component specification.
- **White-box**: The implementation is fully available. Allows interference and implies reusing software fragments. Most class libraries and frameworks are delivered in source forms and application developers study the classes' implementation to understand what a subclass can or has to do.
- **Glass-box**: It can be seen like a limited white-box approach since it allows inspection of the implementation but not interference.
- **Grey-box**: Reveals a controlled part of its implementation.

Independence : If a building block is intended to be considered as a component, should be possible to deploy it independently. The component must be able of working where the current context differ from the original one.

Like every software product, software components have to accomplish a development life cycle, and how Cheesman and Daniels state in their book [CD00], our view of them can change during these stages, namely, requirements, design and provisioning, assembly, deployment, and runtime.

2.2.3 Reuse Mechanisms

After a good definition of a component, we have to state how they can be reused, therefore we need to provide the reuse mechanisms. Terms like composition, interconnection, interaction, communication and interoperation are closely related. Although, in the literature, they have different semantics, they are often used interchangeably. Just to reduce the misunderstanding and clarify the narrative, we will consider just *composition* and *interoperation* with different meaning.

Niestraz and Dami in [ND95] define *component composition* as the process of constructing applications by interconnecting software components through well-defined ways to interact and communicate provided by themselves, that is, *composition rules.*

Composition rules provides the mechanisms that enables prefabricated things to be reused by rearranging them in ever new composites[Szy98].

The ability of software components to communicate and cooperate despite in language, interface and execution platform is called *interoperatibity.* Succesfull composition of components does not necessarily imply their successful interoperation. Two components may simply pass control or send a message to each other. They can also be involved in more complicated form of interoperation, e.g., sharing data or using some component for data access[Sam97].

2.3 Summary

Usually many models that deal with component's description take as starting point fundamental principles that underpin object technologies. However, there are newer approaches, like Schneider and Nierstrasz in [SN99], which make a critic on object orientation respecting with its capability for expressing the architecture.

Until here we have a good idea about the software reuse usefulness, but the doubt may be when in the developing process is suitable to introduce reusing considerations?. The answer is when requirement are available. If we apply a *top-down* approach not always initial user requirements are adequate and consistent with his desires, and system architectures achieved using this technique are specifically adjusted to the initial set of requirements. Thus involved components will suffer the same drawbacks which is in sharp contrast to the idea of building a system from truly reusable components. In the other hand, bottom-up approach starts using reusable components but it becomes impractical for the impossibility of taking the requirements into account early. Consequently a mixture of this popular techniques is convenient when reusing components turns part of development goals.

One thing can be stated with certainty: components are for composition. Therefore components should be understood to be reused. Components have to be designed upon the idea that they will be, reused in different context conditions, and because of that should provide a set of mechanisms which rule the manner of reusing them. In the next section we define a conceptual framework where we enumerate the aspects should be considered to describe a reusable component model.

Chapter 3

Designing Components

Analyzing what software reuse and software component are we may resume that a modular approach is desired rather than a monolithic one. Also, if already exists a solution (well-proven, tested, etc) for the problem in hand then it is better to use that one instead of building another from scratch. It results in reduction of many costs associated with software development (e.g. design solutions, quality testing, coding, etc.). Modularity and reusability intersects in a another concept, *reusable components*.

Reusable component design involves two engineering approaches, *design for reuse*, which considers aspects related with component conception (i.e. identification, representation, organization) and *design by reuse*, which deals with mechanisms to make available the components, either in their original or adapted shape (i.e. searching, adaptation, composition). Considerations about those approaches, for reuse and by reuse, have been described in this chapter.

3.1 Design For Reuse

When we attempt to design a component to be reused it is necessary to provide a framework for addressing the isolation of the fundamental structure of the component, and the structures to support the proper reuse, that means, we need to establish the identification, representation and organization of the component model.

3.1.1 Identification

Component identification let us to get a well understanding of the situation for which the component can be a suitable solution to solve either the whole or part of the problem. The identification can be expressed through the specification of the life cycle, granularity, orientation and range of the component:

- **Life cycle**: Life cycle determine the development phase where the component is applicable in, that is, *analysis*, *design*, and *implementation* components. Most of the existing component models have proven their successfulness and usefulness through their wide use in many applications, but they are mainly components fulfilling implementation level requirements. There are small number of approaches in higher development stages, for example, component specification in design level.

- **Granularity**: Partitioning the software design in components is a subtle process that has a large impact on the success of the resulting components. The granularity unit applied to identify the units that form a system, which is strongly associated with the life cycle phase in consideration. In [Szy98] Szyperski enumerates several criteria for isolating blocks in a system, which eventually can be considered as a component. Traditional units are procedural libraries, classes, and modules. The rules governing the partitioning vary from case to case, hence it is important to understand the implication of the granularity of a particular partition.

- **Product and process orientation**: From certain point of view component-based solution models might be seems as product-oriented one, it means that as result of its use we will get a final product loosing any trace about partial steps. On the other hand, in a process-oriented model the different transformations applied to the original solution belong to the final solution, the resulting component definition is compound by the original component definition plus the *delta* representing the adaptations. In general, in the domain of problem-solving method (PSM), component models meet the process oriented idea since they start with an abstract solution, and then a set of configuration and adaptation rules allows use it to solve a specific problem domain [BPM$^+$98]. To conclude, a trade-off between both orientations could be a suitable strategy because traceability of adaptation makes possible to keep in control evolution issues.

- **Range of components**: We may classify components according to their *specificity*. Thus they might be labelled either *generic*, *domain* or *applications* component. The *specificity* relates the applicability of a component to different domains. For example, considering a components that let us to choose a particular color from a color palette, the range can be bounded for user interface domain and particularly graphical applications. On the other hand, if we have a component that given a grammar specification it let us language parsing, the range for this component seems to be less narrow than the previous example.

3.1.2 Representation

Following its representation a component requires a proper representation. Independently of life cycle, the component should provide all the needed information for understanding the functionality encapsulated into it, even when we consider components as white-boxes (see section 2.2.2). For instance, in many component models interface specification are used to provide this information. An interface description[KBV00] consists of a *signature* part, describing the operations provided and used by a components, and based on that, a *behavior* part, describing the component's dynamic behavior with respect to its interfaces. The meaning of the interfaces as well as the semantic content can vary, for example, when describing a CORBA component only a specification written in an interface definition language is provided. In Eiffel[Mey94] component description add to ordinary interface specification the semantic for reusing component by *features's contracts* (using Eiffel's vocabulary) which include pre-conditions, invariants and post-conditions. Examples for description techniques are graphical notations like class diagrams and state transitions graphs for modelling languages like those included in UML[1], as well as textual notations like provided for IDL[2].

3.1.3 Organization

Components as software artifacts need to be stored in such a way that they can be integrated in a system environment to really accomplish their objective. Providing component implies design a place from where any *consumer* may get components for reuse them, not only local but also remotely. The design of this repository should satisfy the following properties[Mey97]:

- **Easy of use**: when a client decides to use reuse one component should be able to get it quickly. Searching spaces for components have to be structured in such a way that the user might not spend time searching in wrong places.

[1]Unified Modeling Language
[2]Interface Definition Language

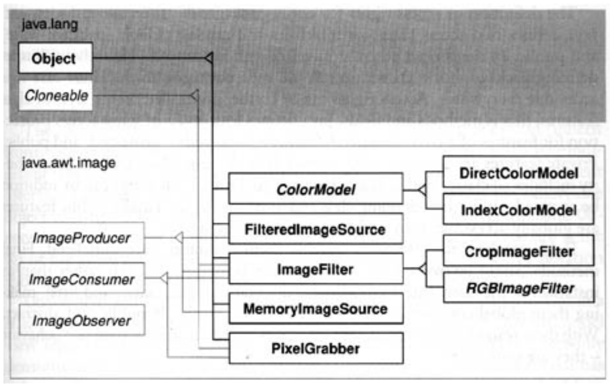


Figure 3.1: The java.awt.image package.

- **Documentation**: The documentation must be accurate and well-organized, so that a user may quickly find the relevant information about a specific component.
- **Extensibility**: The structure design must leave few open doors to allow for its own evolution. It become a problem when we need to preserve the compatibility with existing applications. Version support of the components should be provided aside from the container service.

For instance, a *library* may be considered as the underlying structure to support a repository, in fact, most of the component models use it. And advantage of this well-known mechanism is the possibility to use any searching algorithm available to traverse a tree (see section 3.2). Figure 3.1 shows a java package that represents a hierarchical structure for java classes. Although the library, understood as a tree-based structure, fit for component organization in most cases, should be specified an alternative for components that requires the manipulation of relationships among them more complex, for instance, a lattice-based (e.g. semantic networks). To illustrate an example of lattices, we can use the relationships proposed by Gamma et al.[GHJV94] to connect design patterns (see figure 3.2)and together with the table 3.1 are two way to access to the same component space. Since components can exist at all scales[KS98], is desired that a component-based application be ready for use as a new component, whatever be the organization.

3.2 Design By Reuse

Since a component has been designed for reusing, it should be available for that purpose. Here we describe aspects related with the component availability.

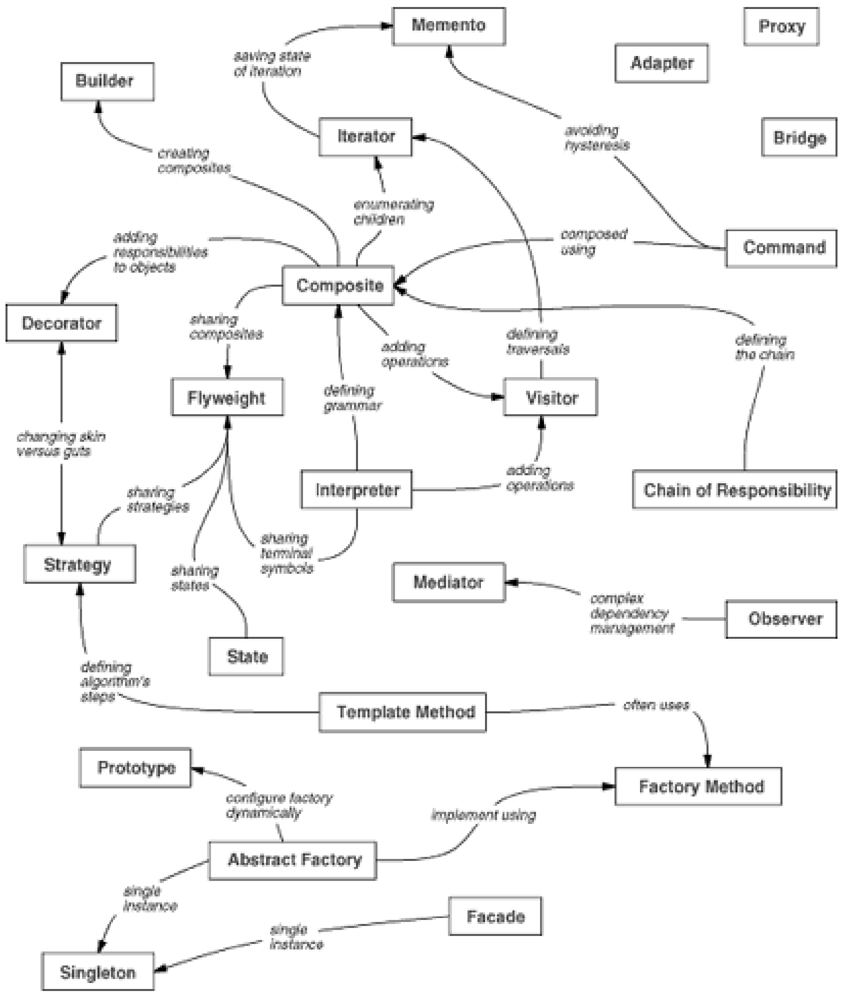


Figure 3.2: Design patterns map.

		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method(107)	Adapter(139)	Interpreter(243) Template Method(325)

	Object	Abstract Factory(87)
		Builder(97)
		Prototype(117)
		Singleton(127)
			Adapter(139)
			
			Bridge(151)
			Composite(163)
			Decorator(175)
			Facade(185)
			Proxy(207)
				Chain of Responsibility(223)
				Command(233)
				Iterator(257)
				Mediator(273)
				Memento(283)
				Flyweight(195)
				Observer(293)
				State(305)
				Strategy(315)
				Visitor(331)

Table 3.1: Design pattern's space proposed by Gamma et al. in [GHJV94].

3.2.1 Searching

Given a problem a solution is intended, then the action to obtain a solution from a repository involves the criteria for searching the most suitable. It is not possible to apply a reusable component that is not easily reachable. The searching process could be affected by the features used to describe the component. Component representation includes basically the elements used for indexing, therefore the success in finding the right component will depend on the proper understanding of the user about the facilities provided by the components. Finally, the organization selected to support the component reuse determines the strategy (algorithm) applied for searching them. Table 3.1 shows the classification made by Gamma et al.[GHJV94] for indexing patterns.

3.2.2 Adaptation

Component may be understood as an entity that involves generic properties, then we need to adapt it to a particular domain. *Adaptation* is the activity of accommodation a component to fit into certain requirements[KS98]. But, what will we really adapt?, it means changes therefore the component model should manage this aspect to control the effects produced by those changes. Many existing models contemplate *adaptation* as the fine tuning performed after a component 'instantiation', that means, you can change a copy of the selected component.

Renaming, creation, deletion and change of elements (e.g. classes, methods, or attributes) allow to satisfy subtleties of a particular domains. Concretely, we can consider as examples of adaptation mechanisms to:

- **Abstract Method**: Through abstract methods we can define partially the behavior of objects represented by this class. A concrete content for this method need to be specified in order to reuse the component to solve the current domain problem.

- **Method overriding**: Although a method has been coded for a particular objective, it is possible to rewrite it for another purpose.

- **Renaming attributes**: The name given for the attribute in the component specification can be a little generic, therefore it may be appropriated adapt these names to the domain specific vocabulary. This increases the expressiveness of the component solution.

- **Inheritance exception**: It is not a common facility provided for object-oriented languages, but it should be useful to decide which elements of the inheritable part of the component specification can be cut off (discarded) from the resulting component.

This is not a closed list of possible *adaptation*. The alternatives will not depend on the underlying programming languages supporting the model, but the rules stated in the model's fundamentals.

3.2.3 Composition

Composition may be defined as the activity that lets us to combine components. The composition of component is performed by using well-defined reuse mechanism that state exactly how a component can be reused in order to build another one. The term *'assembly'* can be used as synonymous of *composition*.

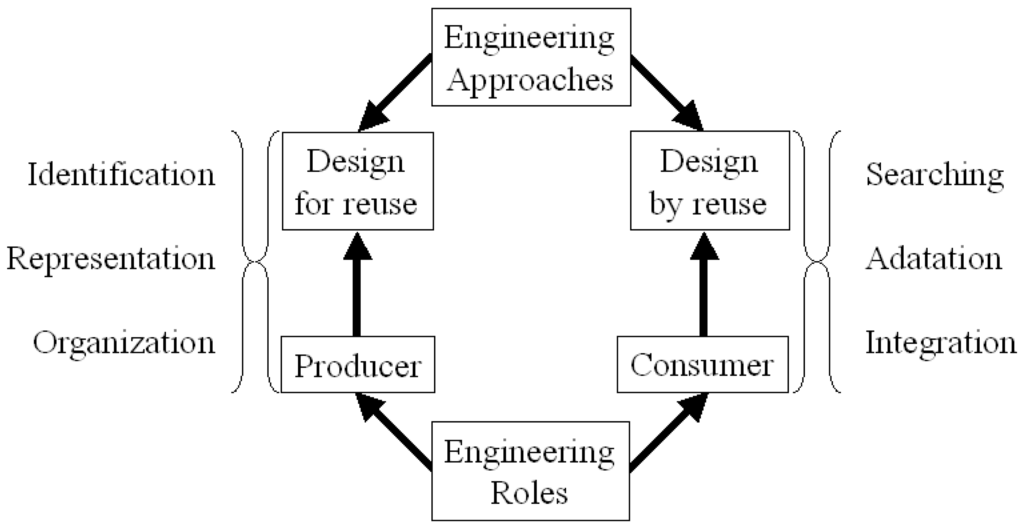


Figure 3.3: Engineering Approaches for designing components.

3.3 Design Roles

Meyer[Mey94] distinguish two users categories or engineering roles that meet to the aforementioned, *reuse consumer* and *reuse producer*(see figure 3.3). Although they are not disjoint, is wanted to stimulate to consumers move a little closer to the producers' side providing a framework where they can be guided to accomplish that. We must keep in mind the existence of them when a component model is built because they may introduce distinct viewpoint in the design. It may sound an evident appreciation, but sometimes the differences affect the basics of the models.

3.4 Describing Component Models

We will use the framework defined in this chapter to describe the existing component models.

3.4.1 JavaBeans

JavaBeans is a portable, platform-independent software component model written in Java. It enables developers to write reusable components. Basically, *beans* are Java classes that can be manipulated in a visual builder tool and composed together into applications[DeS97].

- **Identification**: Beans are expected to support mainly small to medium-sized controls, similar to OLE[3] controls. Hence *JavaBeans* are used to solve *implementation* issues, and the granularity corresponds to the size of the controls above, for example, simple text editors, drawing editor, button constructor, etc..
- **Representation**: Basically a *JavaBeans* could be represented by *events* and *properties*. A property is a attribute that can affect a bean's appearance and behavior. Beans can be announced as potential source or target of specific types of events
- **Organization**: Since the *JavaBeans* follows the same implementation support than Java, the component are organized similarly. They can be stored by any packaging mechanism available for Java implementations, that is namespaces determined by packages, and *Java ARchive*(JAR) files.
- **Searching**: Basically to find a *JavaBeans* we need to locate the corresponding package where it is store. The searching could be facilitated using an assembling tool.
- **Adaptation**: A *JavaBeans* can be customized by settings its properties. Properties may be used by scripting environments, can be acceded by calling *getter* and *setter* methods, or can be acceded using properties sheets at assembly time or runtime.
- **Composition**: The composition of *JavaBeans* is based basically in an event-based communication where the event populated by a *JavaBeans* are the means of composition with other component. Also the attribute manipulation could be a method to integrate components.

3.4.2 COM

COM (Component Object Model) is the Microsoft's foundation on which all components software on its platform is based.

- **Identification**: Since COM is defined as a binary standard, and does not even specify what a component or an object is, its associated life cycle'phase

[3]Object Linking and Embedding

is undoubted the implementation one. The granularity results mainly difficult to define in this case because a COM object just define a standard to follow by any application in the aforementioned planform to become in a component to be reused. For the same reason is impossible to bound the competent range that COM can represent.

- **Representation**: The fundamental entity that represent a component is an *interface.* Internally, this interface represents a pointer to a set of operations available to be dispatched. Those operations could be implemented for any number of classes, and the schema becomes more confuse when a COM component is free to contain implementations for any number of interfaces. A COM component is not necessarily a traditional class and a COM object is not necessarily a traditional object.

- **Organization**: Logical organization in COM involves a categorization in order to support efficient manipulation of a set of interfaces. A COM component can be member of any number of categories and categories are not related among themselves. Physically a COM component belong to a class type which are organized in libraries. Every element in this scenario was assigned an identifier, *interface* (IID), *category* (CATID), and *class* (CLSID).

- **Searching**: A client needs to instantiate a component, the to pick up the right component it has to specify the class of the component to a library. The library executes the procedure to instantiate the selected component. Also it is possible to use the *system registry* available in this platform.

- **Adaptation**: A COM interface cannot be adapted in any way. To support versioning in COM is possible just defining another interface.

- **Composition**:The Microsoft's component model, COM, does not support any form of implementation inheritance which does not mean lack of support for reuse them. COM support two forms of object relationship to enable object be reused, *containment* and *aggregation. Containment* is the simple object composition technique where an object holds an exclusive reference to another. The former, also called outer object, conceptually contains the latter, the inner object(see Figure 3.4). On the other hand, the Figure 3.5 illustrates when the inner and the outer object establish an *aggregation* relationship maintaining mutual references.

3.4.3 CCM

The Object Management Group(OMG) defined CORBA(Common Object Request Broker Architecture) to achieve intraoperatively in the market for objects.

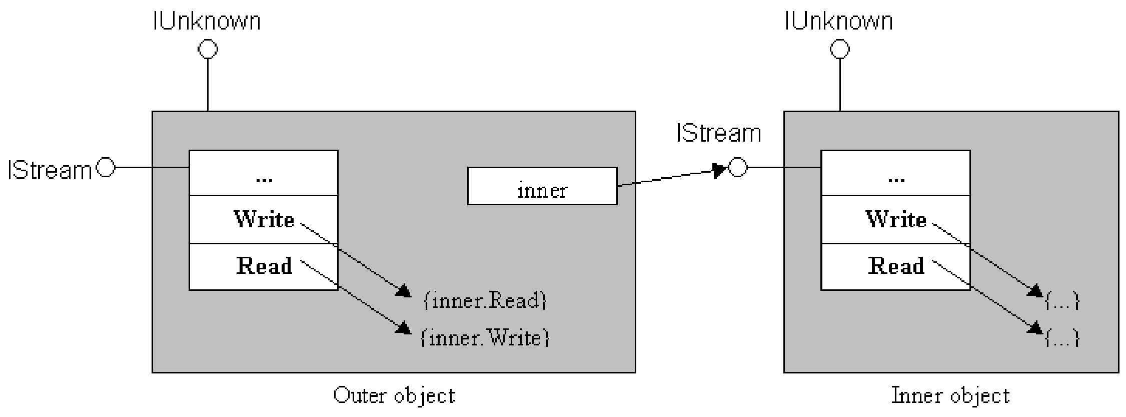


Figure 3.4: COM objects in a *containment* setting.

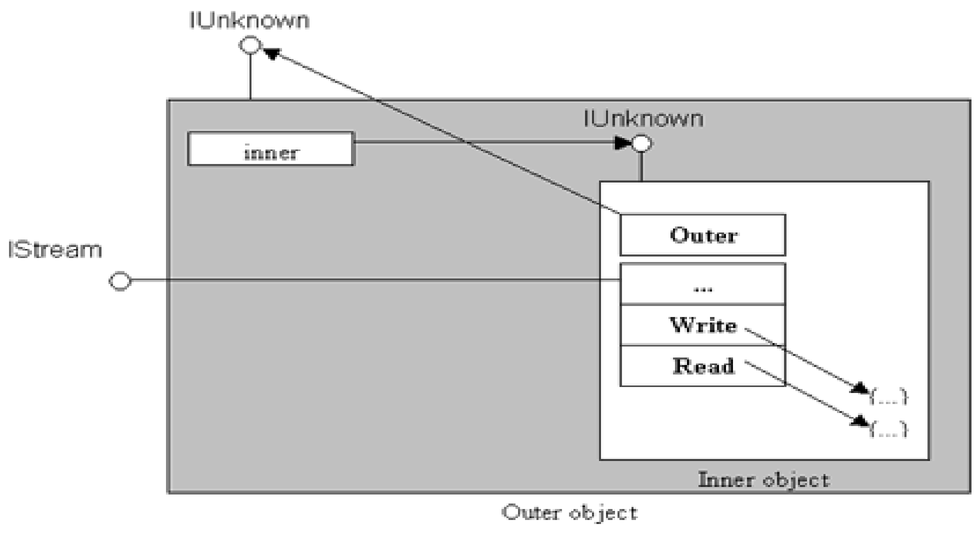


Figure 3.5: COM objects in an *aggregation* setting.

CCM (CORBA Component Model) extends the object model defined in CORBA.

- **Identification**: Since CORBA component is based in CORBA specification we may say this component address implementation issues.
- **Representation**: To express a CORBA component, the IDL (Interface Definition Language) is used. Among the main features of this languages are platform-independency and implementation language independency, aside object-oriented. Since the CCM component extends the CORBA object model, a component could be seen as a meta-type in CORBA, that extends a meta-type 'interface'[Rui00].
- **Organization**: Once interfaces are expressed in IDL, they can be compiled and deposited in an *interface repository.* A repository is supported by an ORB (Object Request Broker), which permits the component communication.
- **Searching**: The searching services is given by ORB repositories.
- **Adaptation**: The component adaptation could be done by component *configurations.* Developers can programm the configuration that permits assign values to the components attributes. Those attribute's values could be defined constrained.
- **Composition**: CCM defines the *receptacles* concepts, which determines union points between components. A receptacle could be seen as requirement exposed by a component that should be satisfied by another one. Another way to integrate components is by using *events.* The *events* establish a producer-consumer an relationship but not directly as is done by *receptacles.*

3.4.4 IBROW3

IBROW3 (Intelligent Brokering Service for Knowledge-Component Reuse on the World Wide Web) is an European project to develop an intelligent brokering service that enables third party knowledge-component reuse through the World-Wide Web [BPM^{+}98]. Figure 3.6 shows the IBROW3 scenario.

- **Identification**: The knowledge components for reuse are problem-solving methods (PSMs) to solve a reasoning problem, such as the design of a technical product.

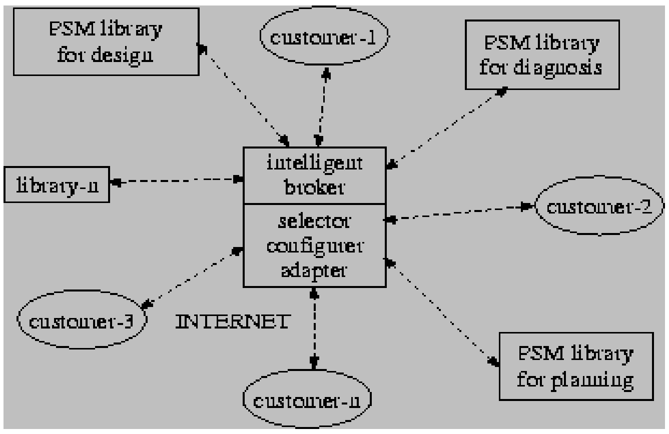


Figure 3.6: IBROW3's scenario.

- **Representation**: For describing component IBROW3 applies a PSM-description language for adequately capturing relevant characteristics, Universal Problem-solving Method description Language (UPML). This language is independent of specific implementations of PSMs, as it is a meta-description defining the broker's view of the PSMs. The formal underpinning of the language are Abstract Data Types, but to exchange the problem-solving methods with other groups a wrapper has to be built. This model defines a KIF[4] wrapper.

- **Organization**: IBROW3 proposes the decomposition of the problem in different, but related, issues. The intelligent broker handles requests for reasoners from various customers. Based on these requests, it accesses different *libraries* available on the Web and searches them for candidate PSMs, which are adapted and integrated into a knowledge system for the customer.

- **Searching**: For selecting PSMs from a library, the broker reasons about characteristics of PSMs, for example about their competence, their requirements, cost of executing a PSM (in terms of execution time or required interaction with a human), its cooperation style (LIFO, LILO), empirical facts of the PSM (how often has it been selected and used successfully), etc.

- **Adaptation**: Since PSMs are generic components, the problem solver has to be adapted to the particular domain knowledge of a customer to obtain a proper knowledge system.

- **Composition**: The broker may need to combine different PSMs that together solve the defined problem. To do this it decomposes the task into subtasks and then looks for PSMs to realize these subtasks, etc. Once the

[4]Knowledge Interchange Format

relevant PSMs are found, the same knowledge can then be used to compose the problem solver.

3.4.5 Design Components

Design components is a research work supported by SPOOL (Spreading Desirable Properties into the Design of Object-Oriented, Large-Scale Software Systems) project [KS98].

- **Identification**: This work introduces the *design component* idea, that is the reification of design pattern in software artifacts.
- **Representation**: The constituents of a design component are similar to those used by Gamma et al. in [GHJV94] to describe design patterns, that is component name and classification, intent, synonyms, motivation, applicability, structure, source code, executable code, etc..
- **Organization**: The components are organized in repositories, from where a client can take a component either local en remotely.
- **Searching**: The components are available via its identifiers, name, classification, intent, etc.
- **Adaptation**: This model permits instantiate component to be customized to meet the specific requirements. The customizations are classified in four dimensions: Scope, Revision, Specificity and Concreteness.
- **Composition**: The composition in this model is achieved by assembling design component.

3.5 Proposed Component Model

In this section we describe the features the model that attempts to solve some anomalies in the existing component model.

- **Identification**: Most of the models described in section 3.4 are model that address implementation level requirements. Implementation-oriented models just provide a means for reusing software by mechanism to assemble components, but they fail when an architectural description is need from the resulting system. We propose to go up in the development level to define a component model for expressing design decision., preserving the underlying architecture.

- **Representation**: The component's representation would express in a generic way the intention of the solution proposed by the component.
- **Organization**: The organization should provide a means to understand in advance for which problem the component is a proper solution, therefore we have to define a good classification structure to achieve this purpose.
- **Searching**: The searching method is defined based on the proposed organization.
- **Adaptation**: The adaptation's schema should define the allowance in the changes that the members of the component could suffer.
- **Composition**: The component description should provide the mechanisms to reuse the components.

3.6 Summary

This chapter defines a conceptual framework to describe and compare reusable component models determining their identification, representation, organization, searching, adaptation, and composition. The fist three cases are components that face component models addressing implementation level requirement. Basically they provide a platform to achieve a component-based development based on the reusability of existing micro-applications, and in many cases more opportunistic-oriented than systemic approach. Unlike COM, CMM and JavaBeans, three well-know models in the component market, the later cases, IBROW3 and SPOOL's model are research works that define another approach, component-based problem solving instead of component-based implementation.

Implementation-oriented models just provide a means for reusing software by mechanism to assemble components, but they fail when an architectural description is need from the resulting system. The lack of this information about design decisions attempt against evolutionary aspects existing in any software development process. Any change in a component-based implementation needs from a deep knowledge about the underlying structure within it.

In the next chapter we define a reusable component model based on the consideration about the proposed model.

Chapter 4

Reusing Design

From the description made in the chapter 3 about the most the component models we have concluded they suffer some anomalies. One of those is the lacks of elements to express the resulting design structure of the application. One approach the address this problem is the design component's approach [KS98] that we will use as stating point to define our model.

4.1 Introduction

Our model is founded in the *component* conception established by Keller et al[KS98]. They state that *design patterns* can be seen as components since:

> ...design patterns package software engineering expertise with domain knowledge into conceptual building blocks upon which more complex and flexible software design can be built.

The aim of our work is contribute with a model able to express the meaning of the design solution provided by a design pattern or any structure that includes a design solution. Let see the spirit of our model illustrated in the figure 4.1. There, it is shown the outlook of components and how they can be reused to form more elaborated design entities. The schema shows three components, *c_1*, *c_2* and *c_3* which are composed by predefined mechanisms, namely rule_c_11 and rule_c_22. In the example, those relationships express the following situations:

A: component *c_2* reuses component *c_1* applying the rule rule_c_11 and

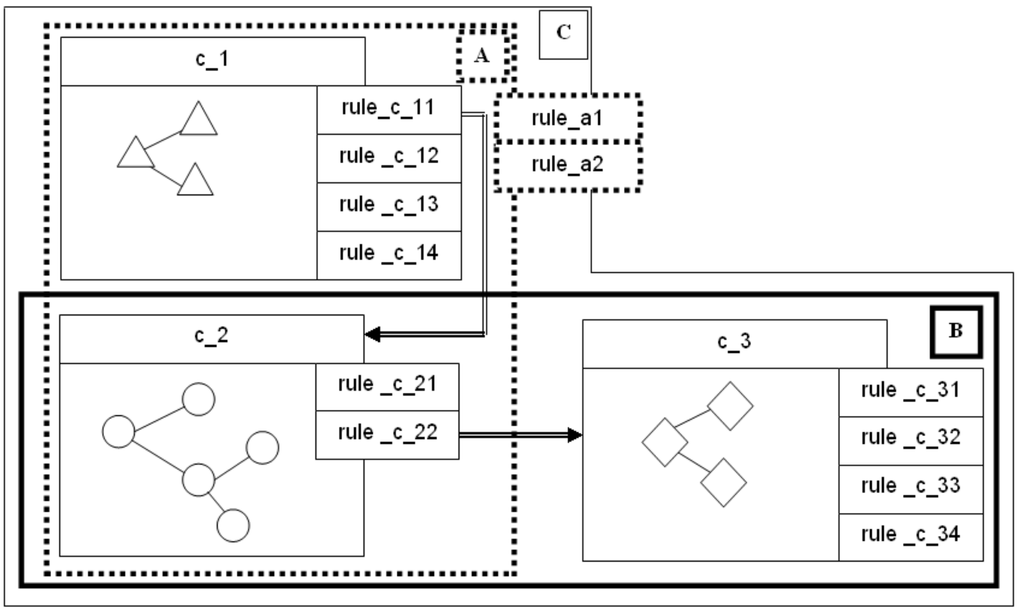


Figure 4.1: Composing design component.

B: component *c_3* reuses component *c_2* applying the rule rule_c_22

Basically what the rule rule_c_11 defines is how the elements forming the solution in *c_1* are involved in this particular reuse process.

Besides to control the unwanted effects in the component reuse, the underlying intention of the model suggested is to supply to the components, information about the composition in which they are included.

Figure 4.1 shows another important feature, the dot line encompassing c_1 and c_2 constitute the boundaries of the resulting component, labelled **A**, from the composition, the same matter is applied to the component **B**. Both components share the same structure that the components, therefore they could contain reuse rules, like rule_a1, and rule_a2.

4.2 ARCoM components

In this section we will describe the proposal, named ARCoM (Another Reusable Component Model) model, using the engineering approaches suggested in chapter

3. Basically we try to pursue the followings:

- To use the object-oriented basis to found our model.
- The resulting model should provide to the components information about the composition in which they are involved.
- To preserve design matters through eventual reusing.

4.2.1 Identification

In this section we make the *identification* of ARCoM components by means the specification of the life cycle, granularity, orientation, and range (see section 3.1.1).

Life cycle : The content of our components are design solutions, therefore the life cycle phase where they may be applied is where design decisions are required. That means, ARCoM components undertake descriptions for the design phase in a development process.

Granularity : How it was stated previously, defining granularity is not an easy assignment. In our case, we deal with two different granularity cases: *producer's* and *consumer's* granularity (see section 3.3), which imply a real difference. For a *consumer* the smaller unit to use for constructing an application will be the solution suggested by the component, so in this case his viewpoint agrees with the component's viewpoint. That sounds obvious since he is the end-user of our system, even when this kind of user is able to observe the implemented solution. But, it does not happen the same with the *producer*, in this case the granularity is focused in the elements used to build the components, that is, classes and relationships. This issue is entirely described in section 4.2.2.

Product or process orientation : Given that the result of every operation between components is an expression of the resulting component, ARCoM model is a product-oriented one.

Range of components : Design patterns describe general design problem that can be applied in a particular context by its customization, therefore an ARCoM component could be classified as *generic-ranged component*, even when the output from customization(adaptation) and integration(composition) be a domain or application component (see sections 4.2.5 and 4.2.6, respectively).

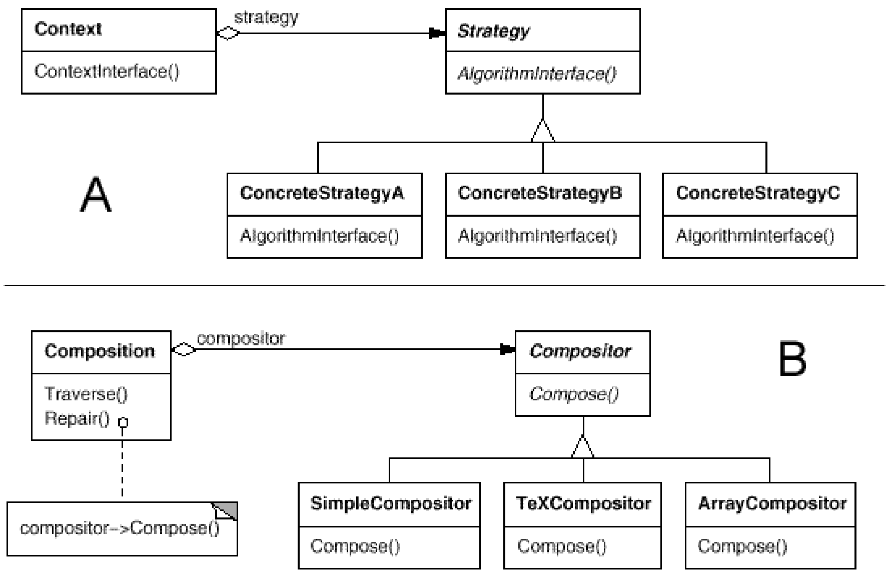


Figure 4.2: Strategy pattern - A) Structure and B) Example.

4.2.2 Representation

Basically, in ARCoM, components are conceived as a set of *properties* and *reuse rules*. The *properties* give the solution's meaning, and the *reuse rules* describe the mechanisms that indeed permit to reuse them. To exemplify the constituents of the component's representation we will use the *Strategy* pattern shown in figure 4.2.

Properties

The properties that set up the *representation* of the components are:

Indexing : The following two attributes determine a first approach of the location where the component may be stored.

- **Name**: The component needs to be identified, then a *name* is assigned. It constitutes the first approach to figure out the solution embedded into the component.
- **Purpose**: It is used as a first level of classification. For example, *creational*, *behavioral* and *structural* suggested by Gamma et al [GHJV94].

Both attributes are used in the component organization when the repository is designed (see section 4.2.3). In the example, **Indexing** part results in:

Indexing = {

Name = `Strategy`

Purpose = `Behavioral` }

Graph : Given a solution, the graph denoted G, that represents the design solution, can be expressed as a set compound by N, the set of nodes (classes), and L, the set of links (relationships) among the elements of N.

$G = \{N,L\}$,

where

$N = \{n_i / n_i \textit{ is a class}\}$,

$L = \{(n_i, n_j, r_j) / n_i, n_j \in N, r_j \in R\}$,

$R = \{r_j / r_j$ is a relationship between the elements in N involved in the solution$\}$(for example: *inheritance*, *composition*, *aggregation instantiation*)

In the example, **Graph** attribute results in:

Graph = {

N = {`Strategy`, `Context`, `ConcreteStrategyA`, `ConcreteStrategyB`, `ConcreteStrategyC`}

L = {(`Strategy`,`Context`,`aggregation`),

(`Strategy`,`ConcreteStrategyA`,`inheritance`),

(`Strategy`,`ConcreteStrategyB`,`inheritance`),

(`Strategy`,`ConcreteStrategyC`,`inheritance`)}

}

Fundamental Properties : **These properties concern to the structure that must remain invariant through any reuse performed on the components.** To schematize this idea we take the *strategy pattern*[GHJV94] that can be described more abstractly than the description given to the client user. Figure 4.2 shows two structures for the pattern suggested. From the difference between pictures **(A)** and **(B)** it is possible to infer that does not matter the number of the *ConcreteStrategy* classes. Only the minimal set, that is, one *Strategy class* (abstract root), one *Concrete class* (concrete child), Context class, an *inheritance relationship* and an *aggregation relationship*, is basically the structure associated to a *strategy pattern.* The *invariants* stated in the *fundamental* properties can be classified in *graph*, *node* and *link invariants.*

Graph Invariants: It is the set of invariants that state which features at graph level should be kept invariant during the eventual reuses performed on the component. Hence they are a set of nodes and relationships:

$$GI = \{N_{GI}, L_{GI}\},$$

where

$$N_{GI} = \{n_i / n_i \in N_{GI}\},$$
$$L_{GI} = \{(n_i, n_j, r_j) / \ (n_i, n_j, r_j) \in L_{GI}\},$$
$$N_{GI} \subseteq N \wedge L_{GI} \subseteq L$$

In the example, **Graph Invariant** attribute results in:

$$\begin{aligned} GI = \{ & \\ & N_{GI} = \{\texttt{Strategy}, \texttt{ConcreteStrategyA}, \texttt{Context}\} \\ & L_{GI} = \{(\texttt{Strategy}, \texttt{ConcreteStrategyA}, \texttt{inheritance}), \\ & \qquad (\texttt{Context},\texttt{Strategy},\texttt{aggregation})\} \\ \} & \end{aligned}$$

Node Invariants: It is the set of invariants that state which features at node level should be kept invariant during the eventual reuses performed on the component. Hence they are a set of expression that define constraints on eventual adaptations on the node's definition:

$$NI = \{(n_i, exp)/n_i \in N_{GI}, \quad exp \text{ is a constraint on } n_i\}$$

Link's Invariants: It is the set of invariants that state which features at link level should be kept invariant during the eventual reuses performed on the component. Hence they are a set of expression that define constraints on eventual adaptations on the link's definition:

$$LI = \{(l_i, exp)/l_i \in L_{GI}, \quad exp \text{ is a constraint on } l_i\}$$

Both, *node* and *link* invariants are associated to the support used to express then, class and relationships. Section 4.2.5 defines how the graph's elements can be adapted.

Macro Invariant Functions: We define two functions to expose the idea that could be possible to associate under a high level concept a set of constraints that may impact on more than one member on the graph and, perhaps, on more than one the aforementioned set of them. Those function take as arguments the graph invariant elements and the nodes that are considered with special features.

- **Context Node**: This operator implies that only the nodes specified as *context* ones could be target of other links (relationship ends) except those that already it has.

  $CN(\text{GI},\text{X}) \longrightarrow NI$

  where

  $X = \{x_i / x_i \in N_{GI}\}$,

  NI= (*it generates a new NI*).

  In the example, we could use this function to set the context class:

  $X = \{$`Strategy`$\}$,

  $CN(\text{GI},\text{X}) = NI \cup \{($`Context`, association=false),

  (`ConcreteStrategy`, association=false)$\}$

  The impact is produced at node level invariants since what the function *CN* should do is to constraint the relationship alternatives of the nodes in N_{GI}, except for the elements in *X*. It returns a new set of *NI*.

- **Root Node**: This operator implies that only the nodes specified as *root* ones could be specialized, that is, be involved in a *inheritance* relationship as parent node.

  $RN(\text{GI},\text{T}) \longrightarrow I$

  where

  $T = \{x_i / x_i \in N_{GI}\}$,

  $I = \{NI, LI\}$.

  In the example, we could use this function to set a Root class:

  $X = \{$`ConcreteStrategy`$\}$,

  $RN(\text{GI},\text{X}) = \{$

$NI = (NI \cup \{(\texttt{Context}, \text{root=false})\}),$
$LI = (LI \cup \{(\texttt{Inheritance}, \text{multiplicity=1})\}$
$\})$

The impact is at node and link level invariants since what the function *RN* should do is to constraint the inheritance alternatives of the nodes in N_{GI}, except for the elements in *X*, and to limit the child number for the inheritance already defined in *GI*.

In this example is more proper define Strategy class as a root node but we choose `ConcreteStrategy` to build a more representative exemplification.

Macro invariants do not belong to the component's representation strictly, they are presented here to introduce the idea that like we generate a set of constraints from a function we could rejoin constraint to express some features of our design.

Specific Properties : Including information about the problem for which the component could be a suitable solution that permits improving the understanding of the pattern's solution. Specific properties involve the idea used by Gamma et al. in [GHJV94] to describe a design pattern. Hence, ARCoM includes as part of the specific properties the followings:

- **Applicability**: It is a text that describes in detail the situations in which the component can be applied for solving the problem.
- **Class Diagram**: The graphical representation defined in UML to communicate the *structural* information.
- **Sequence Diagram**: The graphical representation defined in UML to communicate the *behavioral* information.

Respecting to UML diagrams aforementioned, they are just a initial approach since the other diagrams that deal with other aspects of the modelling could be added in later version of ARCoM. In the example, the specific properties result in:

Specifics = {

Applicability = Use the Strategy pattern when:

- many related classes differ only in their behavior. Strategies provide a way to configure a class with one of many behaviors.

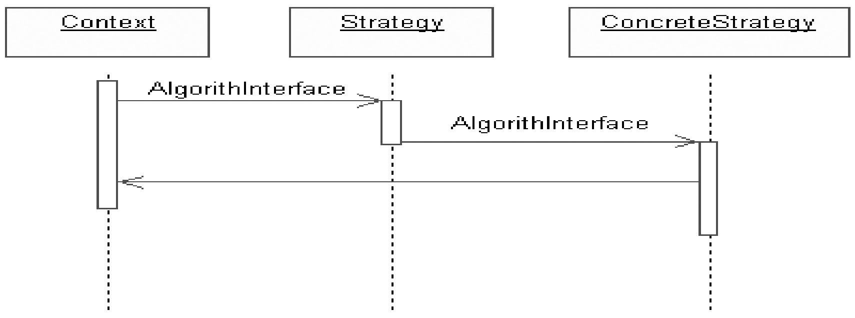


Figure 4.3: Strategy's sequence diagram.

- you need different variants of an algorithm. For example, you might define algorithms reflecting different space/time trade-offs. Strategies can be used when these variants are implemented as a class hierarchy of algorithms.
- an algorithm uses data that clients shouldn't know about. Use the Strategy pattern to avoid exposing complex, algorithm-specific data structures.
- a class defines many behaviors, and these appear as multiple conditional statements in its operations. Instead of many conditionals, move related conditional branches into their own Strategy class.

Class Diagram = (see figure 4.2)

Sequence Diagram (see figure 4.3)}

Reuse Rules

A *reuse rule* defines the effects produced when a given component is reused. It states the manner in which the component properties flow from a component that is reused, named *server-S*, towards a component, named *client-C*. In ARCoM, the reuse rule is defined by the following attributes:

Name : To identify a reuse rule.

Nature : The nature of a reuse rule defines the stiffness of the established relationship between the client and server component. The nature is regulated by two attributes, *dependence* and *predominance*. Figure 4.4 illustrates these variants.

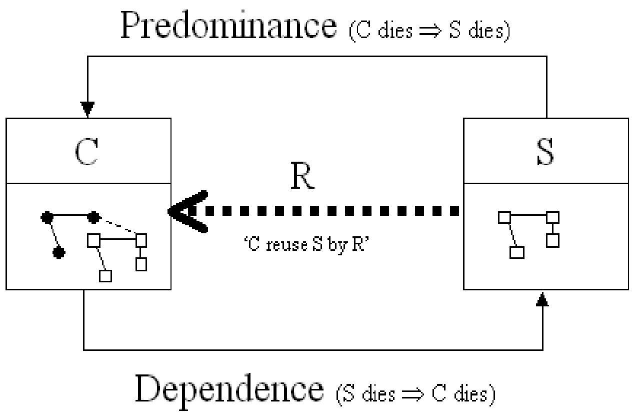


Figure 4.4: Nature of a reuse rule: Dependence or Predominant.

- **Dependence**: A *dependent* relationship implies the client's lifetime depends on the server's lifetime, for example an *inheritance* relationship may be understood as a dependent one between a child member (client) and the parent member (server).
- **Predominance**: A *predominant* relationship implies the server's lifetime depends on the client's lifetime, for example an *composition* relationship may be set as a predominant one between the part (client) and the whole (server) member.

Propagation : The reuse rules allow to customize the manner in which the fundamental properties are propagated in a reuse situation. A reuse rule bases its propagation on two attributes, *Scope* and *Flood*:

- **Scope**: According to the desired effect, the scope can be *complete* or *partial*. In the former case the entire graph flows from the *server* to the *client* component. In the latter case only a subgroup of the graph is transferred to the *client* component.
- **Flood**: It itemizes the graph's elements that are transferred when a *partial* propagation is chosen. Otherwise *all* the elements are propagated.

Method : During the rules' definition it is necessary to set up the method for transferring the elements form the *server* to the *client* component. If it is not needed to preserve a binding with the *server* component, that is, there is a weak relationship stated in the selected rule, the elements are *copied*, otherwise the client keeps a *reference* to the propagated server's elements.

Shareability : This attribute defines the number of possible reuses that a server component can support. It can be *boundless* if there is not any restriction on

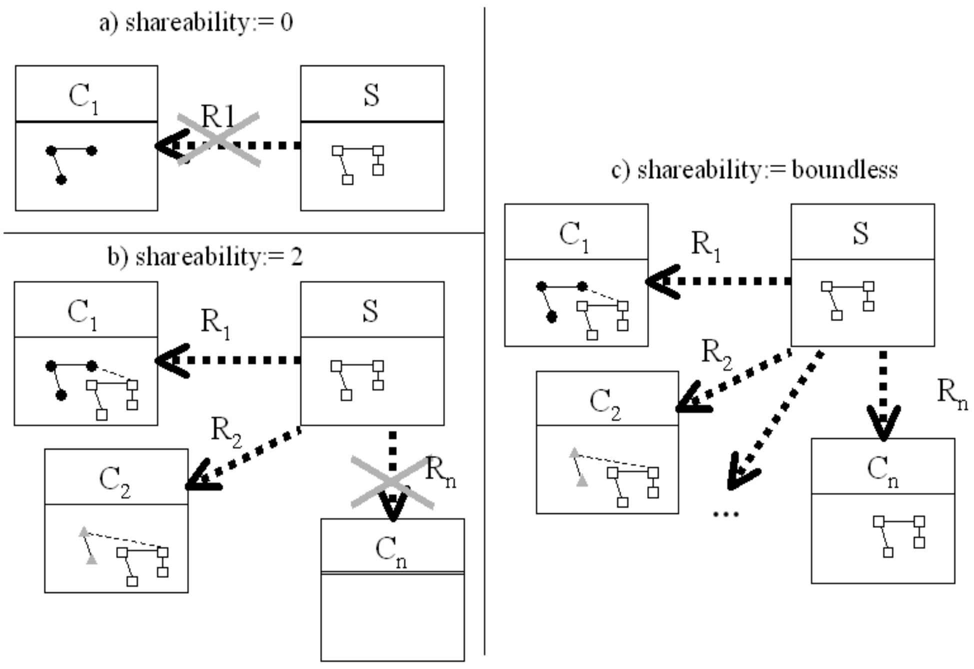


Figure 4.5: Shareability of a reuse rule: a) Shareability not allowed, b)Only allowed for two client components, and c) Allowed for any number of client.

this aspect. Figure 4.5 shows the utility of the *shareability*. This attribute can be understood as the *multiplicity* assigned to the component. Figure 4.6 attempt to depict an example where the shareability should be taken into account. Let suppose that a component that deals with the system security needs to be restricted in order to make sure the manipulation of the overall system security. One way could be to limit the possible reuses to only once.

Transitiveness : When a server component is reused, the definition of the client component is based on its fundamental properties. *Transitiveness* states whether exist, or not, a restriction on subsequent reuses. Figure 4.7 illustrates how the *transitiveness* works. This attribute permits imitate the idea of visibility used in class modelling to restrict the propagation by client component.

4.2.3 Organization

In ARCoM, we distinguish between a logical and a physical organization.

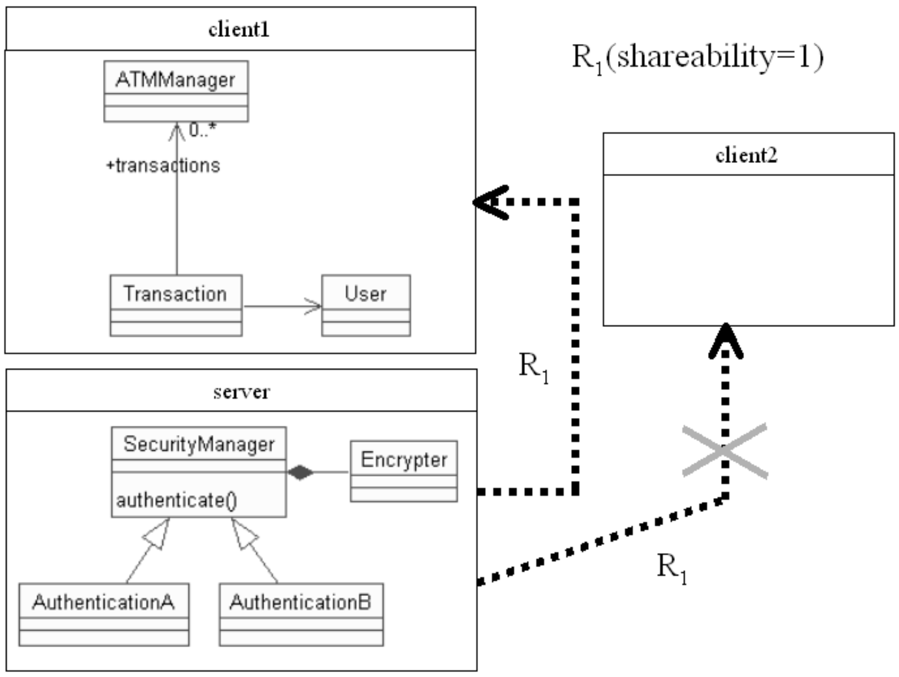


Figure 4.6: Shareability's example.

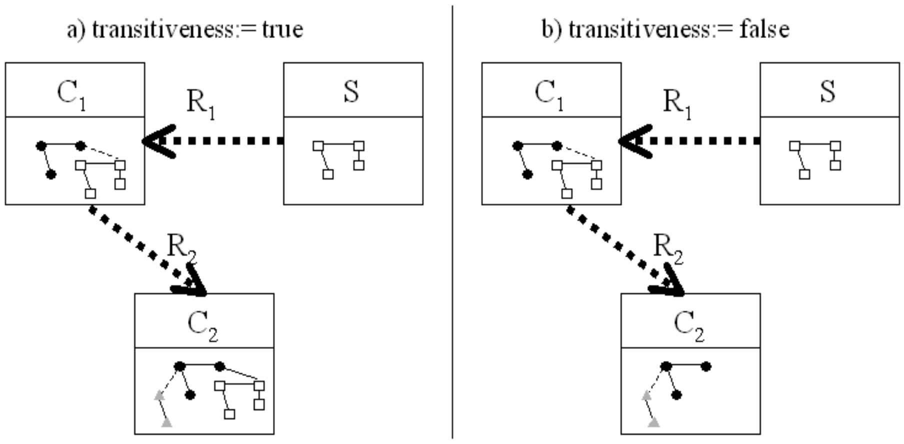


Figure 4.7: Transitiveness of a reuse rule: a)Allowed, b) Not allowed.

COMPONENT

Indexing = {
Name = `String`
Purpose = `String` }

Graph = { **N** = $\{n_1, n_2, ..., n_p\}$ **L** = $\{(n_i, r_j, n_k), n_i, n_k \in N, r \in R\}\}$

Fundamentals = {
Graph Invariants (GI) = {
$N_{GI} = \{n_i / n_i \in N_{GI} \subseteq N\}$
$L_{GI} = \{(n_i, n_j, r_j) /\ (n_i, n_j, r_j) \in L_{GI} \subseteq L\}\}$
Node Invariants (NI) = $\{(n_i, \mathit{exp}) / n_i \in N_{GI},$
exp is a constraint on $n_i\}$
Link Invariants (LI) = $\{(l_i, \mathit{exp}) / l_i \in L_{GI},$
exp is a constraint on $l_i\}$

Specifics = {
Applicability = `String`
Class Diagram = `String`
Sequence Diagram = `String` }

Reuse Rules = {
Rule = {
Name = `String`
Nature = {
Dependence = (true | false)
Predominant = (true | false) }
Propagation = {
Scope = (complete | partial)
Flood = $\{c_i, c_i \in C\}\}$
Method = (copy | reference)
Shareability = (boundless | `Integer`)
Transitiveness = (true | false)}

Table 4.1: ARCom's Representation.

Logical Organization : This aspect deals with the way in which the components will be organized in order to facilitate the component picking for the eventual *consumers*. To organize logically the components, ARCoM uses the *name* and *purpose* attributes. We should consider for later versions a *Related Component* attribute to improve the logical organization.

Physical Organization : Physically, the components are organized in *name spaces*. The smaller space is the component. That is, the elements that constitutes an ARCoM component are associated to a *namespace*. This addressing method permits to trace the elements involved in the reuse processes. Finally, all of the ARCoM components has to be *published* in *directories*, which are the data structures manipulated in the component's repositories.

4.2.4 Searching

Given that the organization proposed in ARCoM involves a logical and physical aspects the component searching obeys the same schema. Logically, we can look for a component through its *name* and *purpose*. After a component has been published by its producer in a directory, it is a responsibility of the *repository manager* (see section 5.2.2) to search the right component in the proper directory.

4.2.5 Adaptation

The classes involved in the solution proposed by the component could be not concrete enough to fulfil the real problem. The *adaptations*, on the component's representation, are governed by the expression assigned to the attribute *Invariants*, that is, all the changes can be performed except when they attempt to violate the constraints imposed by the invariants. The adaptation that we may perform on the graph are classified in three categories, *graph*, *node* and *link* adaptations:

- **Graph Adaptations**: We could add, delete, modify nodes and links.
- **Node Adaptations**: Since nodes are represented by classes, we could add, delete, modify attributes or operation in these classes.
- **Link Adaptation**: If the links are represented by a class we could add, delete, modify attributes or operations in these classes. This separation would permit to map the model in any object-oriented language.

4.2.6 Composition

In ARCoM, the component composition is done by reusing. That is, we will have a composition between components when a client reuses a server one. Given that the reuse rules are used to control the flow between the components involved in the composition, it is important to know how they are evaluated. The entire composition process is explained in section 4.3.

4.3 Composition

Component composition in ARCoM is restricted to one condition, the expression representing the resulting component, form a composition, will be structured in the ARCoM's way, that is it will have *fundamental* and *specific* properties, and *reuse rules* when necessary.

4.3.1 Evaluating a reuse rule

ARCoM's specification includes the definition of an algorithm to be applied in the reuse rules's evaluation. We will analyze the attribute's intervention, one by one, to figure out its real function during the evaluation:

- **Method**: How aforementioned the two values possible to be assigned to this attribute are: *by copy* and *by reference*. By default, ARCoM considers that the transfer way will be done by *copy*.
- **Nature**: The reusing nature is defined by two attributes, *Dependence* and *Predominance*. If some of those characteristics is true implies that the *method* will be set up to 'by reference', since both determine a stronger relationship.
- **Propagation**: So far we have only decided the method but not the range of elements in the graph, on which we will apply this method. For this selection we use *scope* and *flood*.
- **Shareability and Transitiveness**: The *shareability* and *transitiveness* is checked for every element that is included in the *propagation*. That is, if the element it is an original one of the server component, the *shareability* is only checked. But if the server is not the owner of the element we also need to check the *transitiveness* of them.

Inheritance	Composition
Rule = {	**Rule** = {
Name = *"inheritance"*	**Name** = *"composition"*
Nature = {	**Nature** = {
Dependence = true	**Dependence** = true
Predominant = false }	**Predominant** = false }
Propagation = {	**Propagation** = {
Scope = complete	**Scope** = partial
Flood = all }	**Flood** = {`className`}}
Shareability = boundless	**Shareability** = boundless
Transitiveness = true}	**Transitiveness** = true}
Aggregation	Instantiation
Rule = {	**Rule** = {
Name = *"aggregation"*	**Name** = *"instantiation"*
Nature = {	**Nature** = {
Dependence = false	**Dependence** = false
Predominant = false }	**Predominant** = false }
Propagation = {	**Propagation** = {
Scope = partial	**Scope** = complete
Flood = {`someContextClass`} }	**Flood** = all }
Shareability = boundless	**Shareability** = boundless
Transitiveness = true}	**Transitiveness** = true}

Table 4.2: Imitating inheritance and composition by ruse rules

Table 4.8 shows flowchart describing the evaluation process defined in ARCoM. This chart clarify the semantic assigned to the rule's evaluation. By using attributes and evaluation we could model well-known in object-oriented mechanisms to connect objects, for example, *inheritance*, *associations*, etc. Table 4.2 shows four reuse rules that model *inheritance*, *composition*, *aggregation* and *instantiation*, like they are understood in object-oriented programming. For example, in the *Inheritance* rule, the **Nature** is defined as **Dependent** since every child will depend on the parent component, and the propagation is set to a *complete* scope since the father's definition flows entirely to the child's one.

4.3.2 Attributes after composition

In this section we explain the attribute's values resulting form the algorithm explained above. We define an expression for the attributes in the fundamentals. Either specifics and reuse rules in the new component have to be defined manually

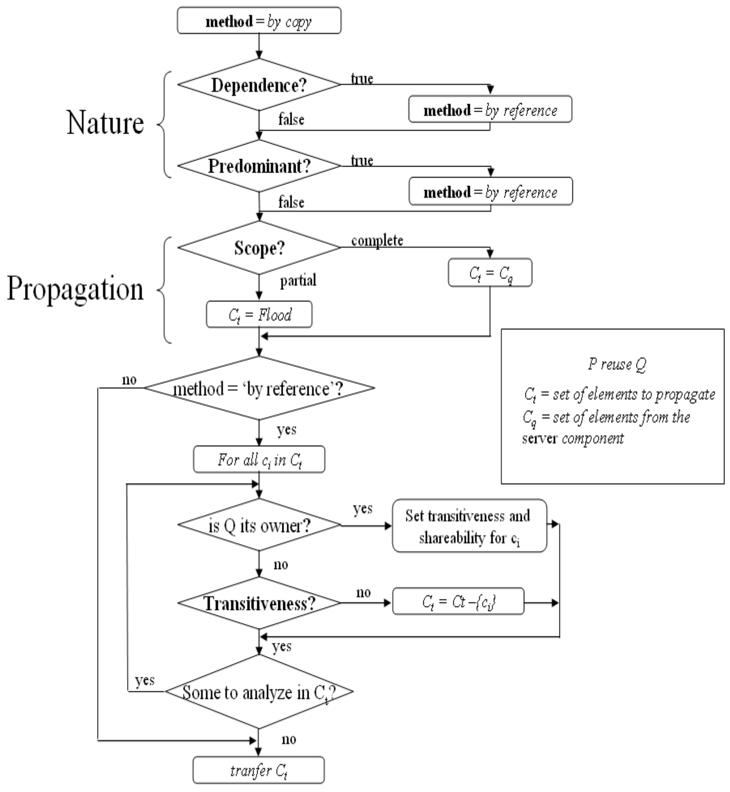


Figure 4.8: Reuse Rule's evaluation (by default).

by the user, that is, *name*, *purpose*, *class diagram*, etc. The ruse rules only have to be defined in the case that the new component need to be reused later.

Given two components, P and Q, both of them expressed in the ARCoM's way, where R_j is a reuse rule defined in Q. We will define the resulting description for the "composition of P and Q by using R_j", denoted $\mathrm{P}{\prec_{R_j}}\mathrm{Q}$, which is equivalent to say that the component P reuses the component Q by applying the reuse rule R_j.

- **Graph**: The set of classes for the resulting component, denoted $\mathrm{N}_{P\prec_{R_j}Q}$, will contain the elements in N_P, classes defined in P, and just the elements in N_Q, classes defined in Q, that are specified by the attribute *Flood* in R_j. That is
$$\mathrm{N}_{P\prec_{R_j}Q} = \mathrm{N}_P \cup (\mathrm{N}_Q \cap \mathrm{Flood}_{R_j}).$$
The set of relationship for the resulting component, that we denote $\mathrm{L}_{P\prec_{R_j}Q}$, will contain the elements in L_P, relationships defined in P, and just the elements in L_Q, relationships defined in Q, whose nodes are included in $\mathrm{N}_{P\prec_{R_j}Q}$.
That is
$$\mathrm{L}_{P\prec_{R_j}Q} = \mathrm{L}_P \cup \{(n_i, r_i, n_k) \in L_Q \wedge n_i, n_k \in N_{P\prec_{R_j}Q}\}$$
- **Graph Invariants**: This set, $\mathrm{GI}_{P\prec_{R_j}Q}$, will contain the graph invariants from component P, denoted GI_P, and Q, denoted GI_Q, but only those related to the elements in $\mathrm{N}_{P\prec_{R_j}Q}$.
$$\mathrm{I}_{P\prec_{R_j}Q} = \mathrm{GI}_P \cup \{(\mathrm{n}_i,\mathrm{n}_j,\mathrm{l}_i) \in \mathrm{L}_{GI} \wedge \mathrm{L}_{GI} \subseteq \mathrm{GI}_Q \wedge$$
$$(\mathrm{n}_i,\mathrm{n}_j) \in \mathrm{N}_{P\prec_{R_j}Q}\}$$
- **Node and Link Invariants**: Idem Graph Invariants described above.

4.3.3 Graph connection

Given that the propagated elements from one component to another also produce the propagation of the invariants that should be preserved in the new component, the connection of the graphs, will be perfomed based on the restriction given for those constraints. That means the client component could relate a server graph element to a client element via any relationship that does not generate any conflict in the state defined by the invariants. If one node could no be inherited no any other element can be child of it, for instance

4.3.4 Composition's commitment

If a deeper analysis is done on the *Nature*, *Shareability* and *Transitiveness* attributes we can resume that they provide a means to proclaim the engagement between the components. For instance, the *Nature* of a reuse rule represents a first approach for a commitment between the component involved. That means, none of them ignores the fact that is part of a composition. The same analysis may be applied to the other two rule's attributes, that is, *Shareability* and *Transitiveness.* Those attributes have an active participation during the evaluation of the flow effects produced for applying a rule.

4.3.5 Customizable evaluation

As aforementioned, ARCoM supplies a predefined evaluation method of a reuse rule. Figure 4.8 shows a flowchart with steps performed to evaluate a *reuse rule.* What the picture shows is one way to evaluate the attribute's configuration, that means, given a component we could change the evaluation in order to give to the reuse rule the desired meaning. In other words, the effect of a *reuse rule* could be changed not only through the attribute's value (like a set of flags that can be set according to the flow suggested in the figure), but also by 'overriding' the existing evaluation method.

4.4 An Example

Let suppose that we take the design pattern named *bridge*, to exemplify the model (see figure 4.9). The result of applying our model to describe this pattern is shown in the table 4.3.

The indexing attributes, *Name* and *Purpose*, fit the same classification used in [GHJV94].

The class *graph* considers the classes and the relationships included the pattern's definition. From the UML model depicted in figure 4.9 we can define:

Graph = {

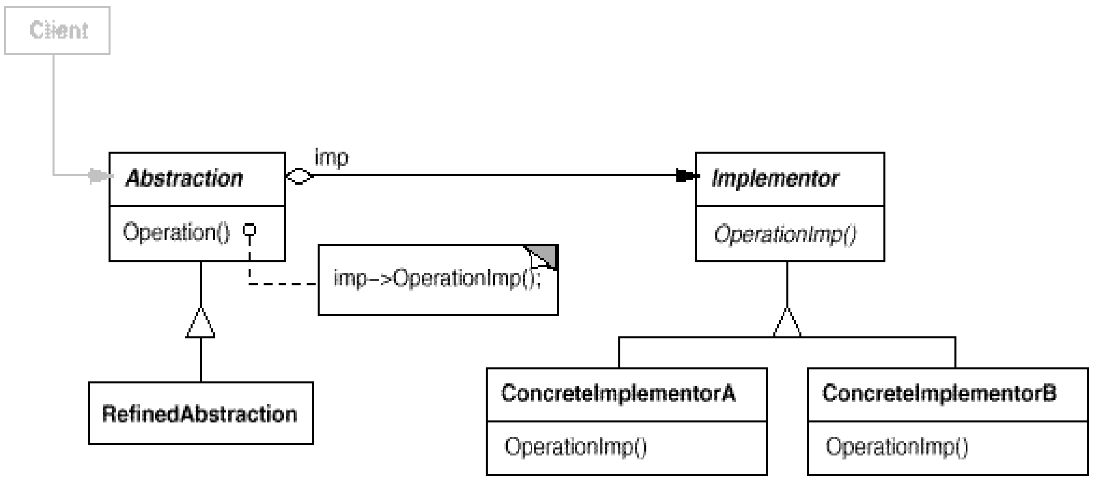


Figure 4.9: Design pattern: bridge.

$\mathbf{N} = \{$`Abstraction`, `RefinedAbstraction`,

`Implementor`, `ConcreteImplementationA`, `ConcreteImplementationB`$\}$

$\mathbf{L} = \{($`Abstraction`,`RefinedAbstraction`,`inheritance`$)$,

$($`Implementor`,`ConcreteImplementationA`,`inheritance`$)$,

$($`Implementor`,`ConcreteImplementationB`,`inheritance`$)$,

$($`Abstraction`,`Implementor`,`aggregation`$)\}$

$\}$

In the *Fundamental* properties we start to decide how the component will be reused in the future. Then the invariants are:

$GI = \{$

$N_{GI} = \{$`Abstraction`, `RefinedAbstraction`,

`Implementor`, `ConcreteImplementationA`$\}$

$L_{GI} = \{($`Abstraction`,`RefinedAbstraction`,`inheritance`$)$,

$($`Implementor`,`ConcreteImplementationA`,`inheritance`$)$, (`Abstraction`,`Implementor`,`aggregation`)}

}

We could use the invariants functions, for example we may define `Abstraction` as a *Context* class, and Implementor as *Root* class:

CN(GI,`Abstraction`) = NI ∪ {(`Implementor`, association=false),

(`RefinedAbstraction`, association=false),

(`ConcreteImplementorA`, association=false)}

RN(GI,`Implementor`) = {

NI = (NI ∪ {(`ConcreteImplementorA`, root=false)}),

LI = (LI ∪ {(`Abstraction`,`RefinedAbstraction`,(`Inheritance`),

multiplicity=1)})

}

The Specific properties involve information that enhances the component understanding, therefore the model includes textual explanations about the applicability of the component, UML diagrams, like *Class* and *Sequence* diagrams, as well.

For this example we define only two reuse rules, *implementation-inheritance* and *aggregation-comprise*, but obviously it is not a closed list. The number of rules is limited for the all possible combinations of the attribute's values.

Firstly, the *implementation-inheritance* rule is defined as a *dependency* nature mechanism since a client component(child) that reuse this server component(parent) will depends on its lifetime. Nothing happens with the server component if a client dies, then that relationship is not *predominant.* By implementing the classical inheritance definition, *all* the features flow from the parent component to the child component in a *complete* way. That issue is addressed by *Scope* and *Flood* attributes with *complete* and *all* values respectively. They are been bound to *boundless* and *true* values respectively. The assembling between components could be made either by `Abstraction` since it is the unique element in *Context Classes* attribute.

And finally, the *aggregation-comprise* rule differs, from the previous one, in the *nature*, it is weaker since both attributes were bound to false values. Appendix C includes a concrete example.

4.5 Comparison

ARCoM model differs from the Keller et al. [KS98] that it approach permits express more generic design solutions. But, perhaps the most important distinction is the ARCoM model allows to express composition information. When a component is part of a bigger solution knows about the mechanism applied to be reused, and aside the component's elements propagation, it includes attributes that enrich the composition process, for instance *Shareability* and *Transitiveness*. Finally, the *ruse rules* are based on well-known concept, copied form object-oriented paradigm, for example *inheritance*, *associations*, etc.

4.6 Summary

ARCoM states that a component may be represented by a set of attributes that describe the picture of the design solution intended, these attributes are divided in four groups, Graph, Fundamentals, Specifics and Reuse rules. There is an standard way to evaluate the rules, that could be redefined as wished. The reuse rules may be, metaphorically speaking, compared to a electronic table where the switches are connected by following a determined configuration. Then we can change the behavior of the table, not only altering the switch position but also modifying the connection behind. The relation between switches is the semantic defined for the evaluation of the attributes conforming a rule (see figure 4.8). We could consider ARCoM components as grey-box oriented since they could be intersected but restricted by the defined invariants.

```
COMPONENT
--------------------------------------------------------------
Indexing = {
        Name = "bridge"
        Purpose = "behaviorial" }
--------------------------------------------------------------
Fundamentals = {
        Graph = {
          ... see above
        }
--------------------------------------------------------------
Specifics = {
        Applicability = "Use the Bridge pattern when:
           - you want to avoid a permanent binding between an abstraction
          and its implementation. This might be the case, for example, when
          the implementation must be selected or switched at run-time.
           ..."
        Class Diagram = class_diagram_file_reference
        Sequence Diagram = sequence_diagram_file_reference}
--------------------------------------------------------------
Reuse Rules = {
          Rule = {
            Name = "implementation-inheritance"
            Nature = {
              Dependence = true
              Predominant = false }
            Propagation = {
              Scope = complete
              Flood = all }
            Shareability = boundless
            Transitiveness = true}
          Rule = {
            Name = "aggregation-comprise"
            Nature = {
              Dependence = false
              Predominant = false }
            Propagation = {
              Scope = complete
              Flood = all }
            Shareability = boundless
            Transitiveness = true}
```

Table 4.3: Bridge pattern's representation.

Chapter 5

Supporting ARCoM Components

To support components produced under ARCoM's specifications, there are some tools that might be supplied to give a guarantee that those component will be ARCoM-complaints. In this section we describe, firstly, our model using MOF[1] to get a complete idea about it, and secondly, we propose a family of tools that allows, to both user profiles *consumer* and *producer*, achieve the manipulation of ARCoM components. Appendix A includes a description in details about the prototype's implementation.

5.1 Meta Model

Figure 5.1 shows the meta model of ARCoM specifications. In this diagram we may distinguish the structure for the design components defined in ARCoM, that is, *FundamentalProperty*, *SpecificProperty* and *ReuseRule*. Another feature included in the model's diagram is the *NameSpace* concept used in the component composition.

5.2 Implementation schema

To manipulate ARCoM components we design a schema where a family of tool deals with all of the activities that conforms the engineering approaches defined in chapter 3. Figure 5.2 shows this schema.

[1]Meta Object Facility

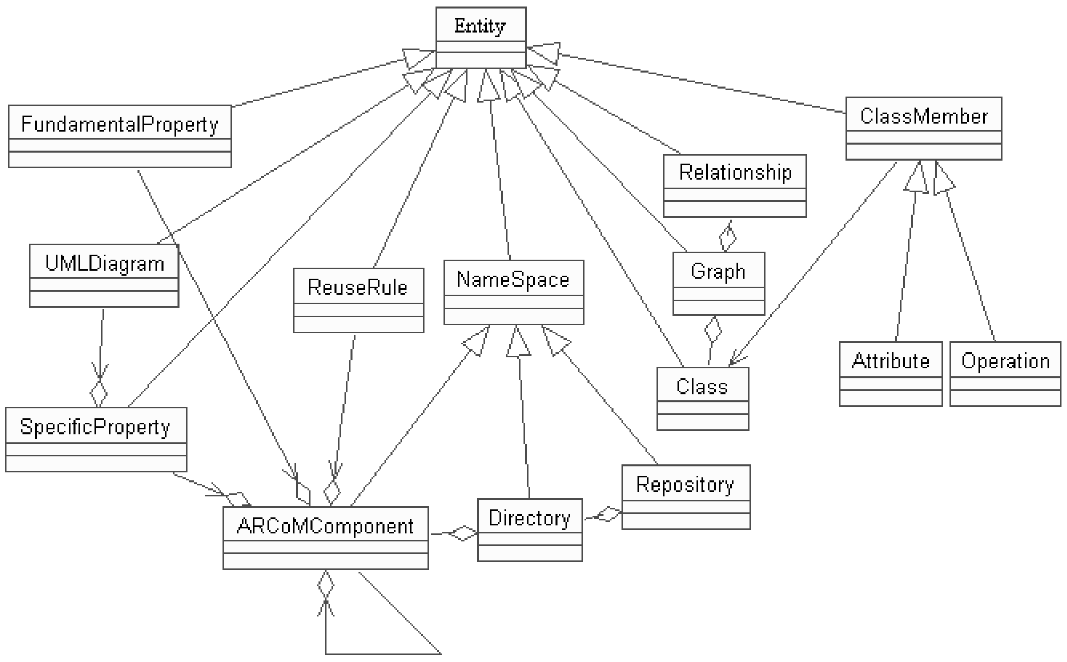


Figure 5.1: ARCoM's MOF-based diagram.

According to the *producer* and *consumer* viewpoints (see section 3.3), the family of tools is divided in two groups: *Component Producer Manager (CPM)* that permits to the producer generates a component, and *Component Consumer Manager (CCM)* that deals with the component reuse. They make use of three other more specific tools, *UML Editor*, *Constraint Manager*, and *Repository Manager*, which provide a means to specify the content of the representation suggested for ARCoM (see figure 4.1). In the following two sections we will describe the functions the tools should cover for supporting the component management. Implementation issues are depicted in Apendix A.

5.2.1 Component Producer Manager - CPM

As stated before the user that interacts with the *CPM* is the *producer*. The *CPM* permits obtain the component *representation*, that is a ARCoM-compliant one, therefore this tool provides an interface from where the user determine the values for the attributes. The schema considers the usage of the *UML Editor* to generate the model from we will extract, by using this graphical interface, the information concerned with the fundamentals properties (i.e. class graph), and the *Constraint Manager* for the specification and checking the invariant expressions. And finally, once the representation is specified enterally we desire store the component in a

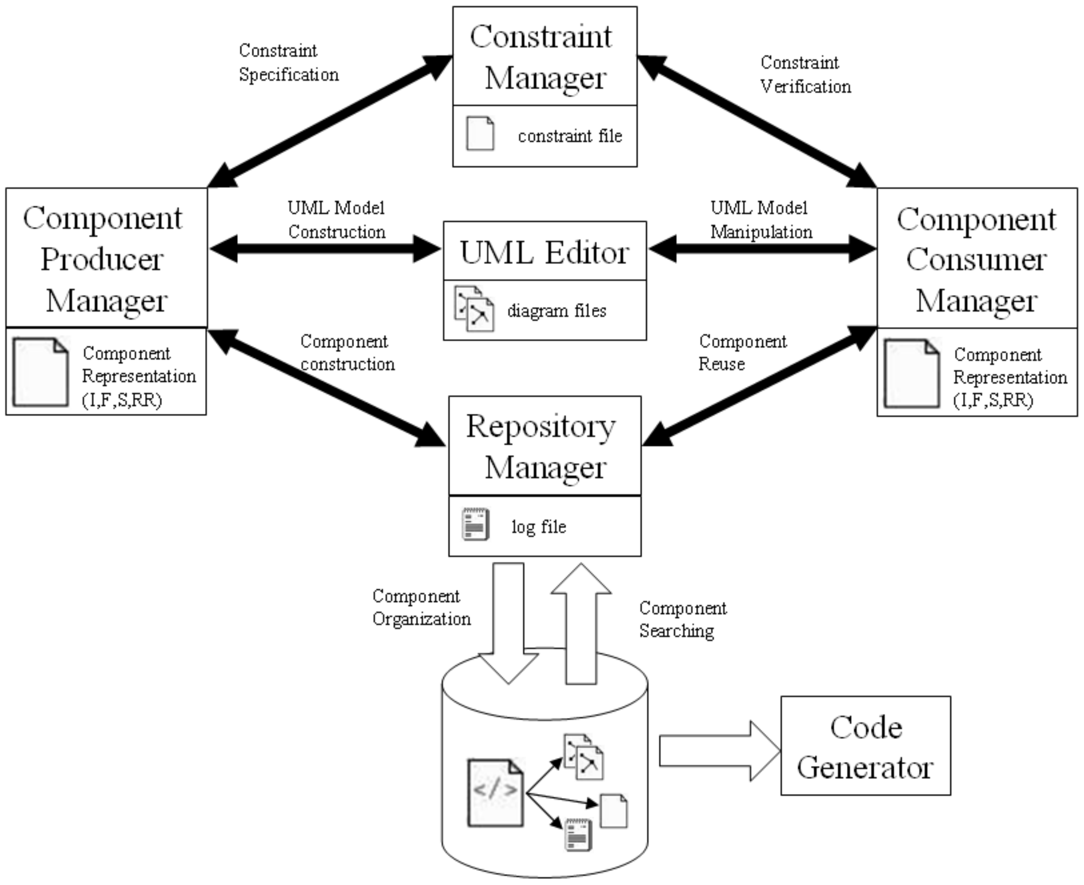


Figure 5.2: ARCoM's tool family.

place where it is available for the consumers, the *repository*.

- **An UML Editor for drawing models**: The UML Editor let to the user not only define the UML-based diagrams comprised in the ARCoM components, but also to use the graphical interface to determine the elements for the *Class Graph*, *Context Class*, etc.
- **Constraint manager for Invariants**: *Fundamental* properties are attributes for which it is needed a well-formed expression that will be used for later checking, for this kind of attributes this tool should provide the mechanism to analyze those expressions. In those cases, since the invariants will be written in Object Constraint Language [WK98], a parser for this language will be used (see section A).
- **A Repository to broadcast a component**: As the main output, the *CPM* generates a component's file the according to the ARCoM's representation. To conclude the circuit, the component has to be *published* to be reused by a consumer user. The place where the components will be stores is name *repository*, and basically provides a *directory* service, that is, an index through which a component can be looked up. The index criteria proposed states the utilization of the *Purpose* attribute as the fist level of classification and the *Name* attribute as the second one.

5.2.2 Component Consumer Manager - CCM

The user profile that fits with the *CCM*'s functions is the *producer* one.

- **Picking up components**: The *CCM* supplies the required functions for reusing the components that the consumer picks up from the *repository*. At this moment the *CCM* applies a searching strategy according to the indexing one.
- **An UML Editor for design by reuse**: The editor offers an interface to generate a design solution by reusing the components through the available reuse rules. But the most important function is that associated to the triggering of the reuse rule selected in the composition.
- **Constraint Manager: Ensuring consistent invariants**: The Constraint Manager has a great responsibility when a component composition is produced. Mainly, it has to check that the resulting set of invariants remains in a consistent state, that is, none contradiction appear among them. Another job is the calculation of the set of invariants based on the propagation effects.

5.2.3 Code generation

ARCoM component's representation involves information about the classes included in the component enough to obtain the corresponding expression in some object-oriented language. Therefore we need to define the transformation rules similar to any design tool that manipulate XMI files to express a model, for instance.

5.3 Summary

We define the family needed to work with ARCoM components based on the user roles (see section 3.3), *producer* and *consumer*. This division permits clarify the scenario where the component could live.

Although in the schema to support ARCoM components we made some consideration about graphical user interfaces, it should be advantageous design the manipulation libraries independently of the interface used to facilitate the component manipulation (see section A.2.6).

Chapter 6

Future Works

6.1 Model

6.1.1 Model Formalization

Although we tried a formal definition for the fundamental properties, particularly for the result from *component composition*, this formal approach it is just that, the beginning. We could need a more robust formal description of our model not only to prevent ambiguous concepts, but also to provide the necessary framework to demonstrate some useful properties. For example, we might need to know whether our model behaves like the types in object-oriented programming, that is, to check substitutivity between components, or to get the relationship between the sequentiality of the reuse rules.

6.1.2 Separations of Concerns

Since ARCoM model can be seem as a pattern-oriented programming approach, the patterns could be categorized in such a way that each of those category addresses a specific concern, namely, propagation, transportation, and synchronization patterns[HL95], for instance.

6.1.3 Organization

The current version of ARCoM introduces a component's organization rather poor that fails when a non abstract pattern need to be located in the structure. For example, the modelling of an editor can be understood as a concretization of the *observer pattern*[GHJV94], and although this *editor* component has an observer conception, this classification schema is not enough for some specific case. It should be considered, as an improvement in our model, to introduce a similar concept as the *semantic network*[ASC98], which are broadly used in *Artificial Intelligence*. Semantic networks would provide the necessary support to express relationships among components more *semantically* reach than that provided by a tree.

6.2 Implementation

6.2.1 Extending Constraint Language

Although the OCL provide a well-proven support for constraint specification, for example at meta-level, some invariants concerned with the meaning of the pattern are not simple to express. For that reason, we consider an extension in the corresponding grammar specification to introduce structures to achieve more natural expression.

6.2.2 Graphical Notation

Finally, we will need to define a graphical expression for ARCoM components to differ them from another graphical meanings. In terms of UML we need to use the stereotype mechanism to achieve a graphical solution UML-compliant.

Chapter 7

Conclusions

This thesis describes a model and the corresponding supporting tool in order to design the reuse in component-based systems. The resulting tools is considered as a methodological guide since it obey to the engineering approaches described for component design. On the one hand, design *for reuse* address the identification, representation and organization of components. On the other hand, design *for reuse* deals with searching, adaptation and composition components.

The prototype allows to show the interest and the feasibility of such approach. Our result is presented as a kind of methodological design approach organized in three steps: Component Producer Management, Component Consumer Management and Code Generation. ARCoM is also exposed as a representation model to design components and composition in three parts: *fundamental* and *specific* properties, and *reuse rules.* Basically a the properties could be seen as the structure and the rules as the behavior of the component when it is reused.

The suggested model does not consider its application in a reverse engineering context, that is, there is not a support a pattern matching in order to obtain the underlying architecture of a given application.

Appendix A

Implementation Issues

A.1 ARCoM components in XML files

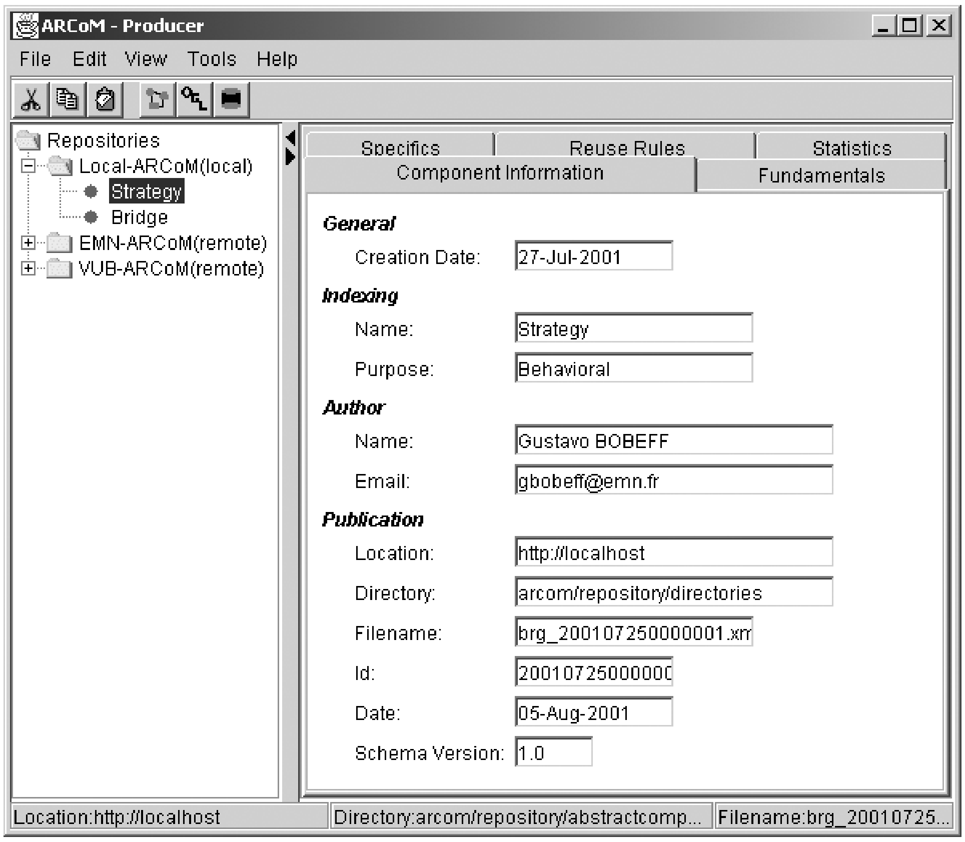
The selection of the most suitable file format for storing a ARCoM component was not a complex task. There are many attributes for which XML[1][W3C00] seems to provide the best solution, namely simpleness, flexibility, extensibility, meta-model neutral, programming language neutral, API streamable, textual, human readable. But perhaps the most useful feature of this tagged format, designed for information interchange, is the ability to separate data and metadata, representation and content.

Particularly, for our work the ARCoM's specifications should be understood, in terms of XML technology, as the grammar to which a ARCoM component has to fulfil in order to be considered a compliant one. For all those reasons we use the XML technology, particularly the related to the *schemas*,[2][W3C01] to implement the component representation.

For our work we define two schemas files, to store components and repository management information, respectively. In the former case the structure in the schema satisfies mainly the ARCoM's representation depicted in table 4.1. Few inclusions have been made not only to enrich the component information (e.g. publication, subscriptions), but also to solve implementation issues (e.g. *namespaces* manipulation). In the Appendix B we explain in detail those files.

[1]eXtensible Markup Language

[2]Another alterative to determine the data structure is to use a DTD (Data Type Definition) instead of schema files.



Repositories
- Local-ARCoM(local)
  - Strategy
  - Bridge
- EMN-ARCoM(remote)
- VUB-ARCoM(remote)

General

Creation Date:	27-Jul-2001

Indexing

Name:	Strategy
Purpose:	Behavioral

Author

Name:	Gustavo BOBEFF
Email:	gbobeff@emn.fr

Publication

Location:	http://localhost
Directory:	arcom/repository/directories
Filename:	brg_200107250000001.xm
Id:	20010725000000
Date:	05-Aug-2001
Schema Version:	1.0

Location:http://localhost | Directory:arcom/repository/abstractcomp... | Filename:brg_20010725...

Figure A.1: ARCoM-Producer's user-interface.

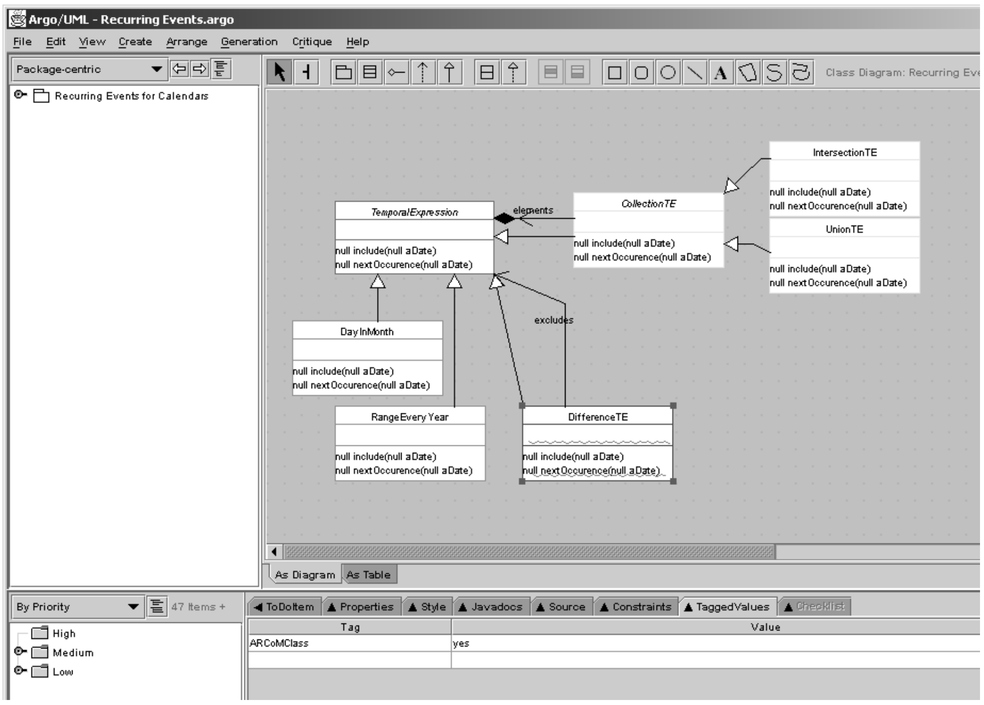
A.2 Application

In this section we describe a prototype of the tool family to support the manipulation of ARCoM component, taking as a guide the schema proposed in section 5.2.

A.2.1 Producer tool

From this application it is possible to build a ARCoM component for reused. Figure A.1 shows a snapshot of the *ARCoM-Producer* tool.

Through this interface we can browse the many repositories available where the component are published, and therefore be eventually picked-up to be reused. There are remote and local repositories. The local one works as a temporal repository where the local producer user store the component not available for be reused yet. Basically, form this application is possible to:



Tag	Value
ARCoMClass	yes

Figure A.2: UML Editor's user-interface.

- Browse the repositories on which the user has the privileges to do this.
- Build a component from the scratch. If the producer need to reuse a component to build another one should use the *ARCoM-Consumer* tool (see section A.2.5).
- Define reuse rules for component from composition.

The output of this tool is a XML file (see Appendix B)

To specify some attributes, the *ARCoM-Producer* tool uses the UML Editor, Constraint Editor and Repository Manager.

A.2.2 UML Editor

This application not only provide a means to draw UML diagrams but also to specify a value for the attributes that involve a reference to members of the class graph. Figure A.2 shows the UML Editor used in the prototype.

We use a tool, named ArgoUML (see below section A.2.6), as the UML Editor. We apply the *tag*[BRJ98] facilities provided by the UML's specification to indicate which of the class diagram's members belong to the *Graph*, *Context Class*, *Dominant Class*, etc.. Then, this editor permits:

- To create the UML diagrams associated to the ARCoM representation.
- To specify the elements involved in the fundamentals.
- To provide the graphical interface for the composition process in the *ARCoM-Consumer* tool (see section A.2.5).

Once determined the *tagged* elements, the ARCoM-Producer makes the parsing of the XMI[3][Kum00] files, generated by the editor, to extract the information.

A.2.3 Constraint Editor

To determine the attributes that imply constraints, namely Graph, Node and Links Invariants, we use a Constraint Editor. Figure A.3 shows the user interface of this editor.Table A.1 enumerates the adaptation the elements in the solution may suffer.

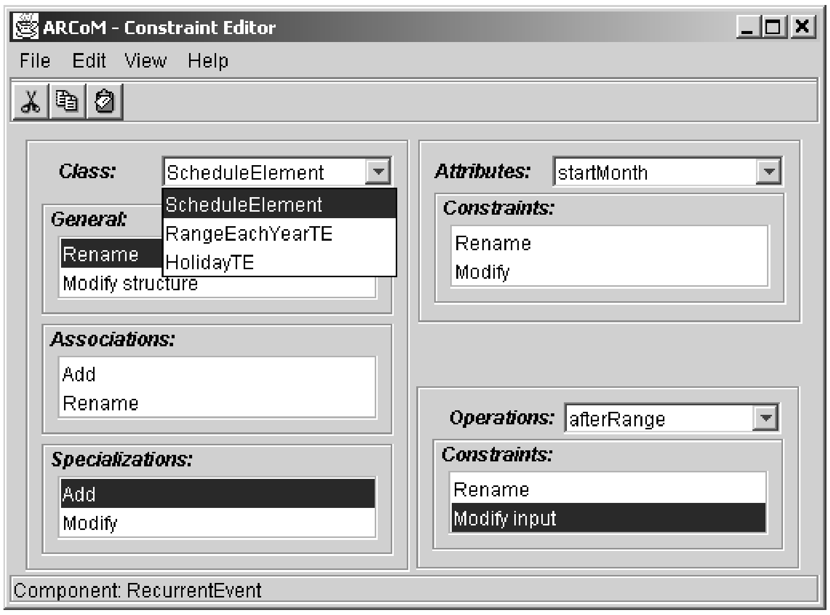
Because of the selected constraint language, all the constraint that we can determine in the graph's elements can be translated in constraint on the classes (nodes). That because we introduce the link's information inside the classes. In order to provide *traceability*, the *adaptations* produced in a component will be logged preserving references to the original elements.

A.2.4 Repository Manager

The ARCoM-Repository Manager services to both user roles. The *producer* can organize the repository, and the *consumer* localize a particular component to reuse. Figure A.4 shows the user interface implemented for the prototype. Among the services provided by this tools are:

- To browse the directories of components accessed from the repositories.
- To publish components through the directories in repositories.

[3]Metadata Interchange Format



Class: ScheduleElement

Attributes: startMonth

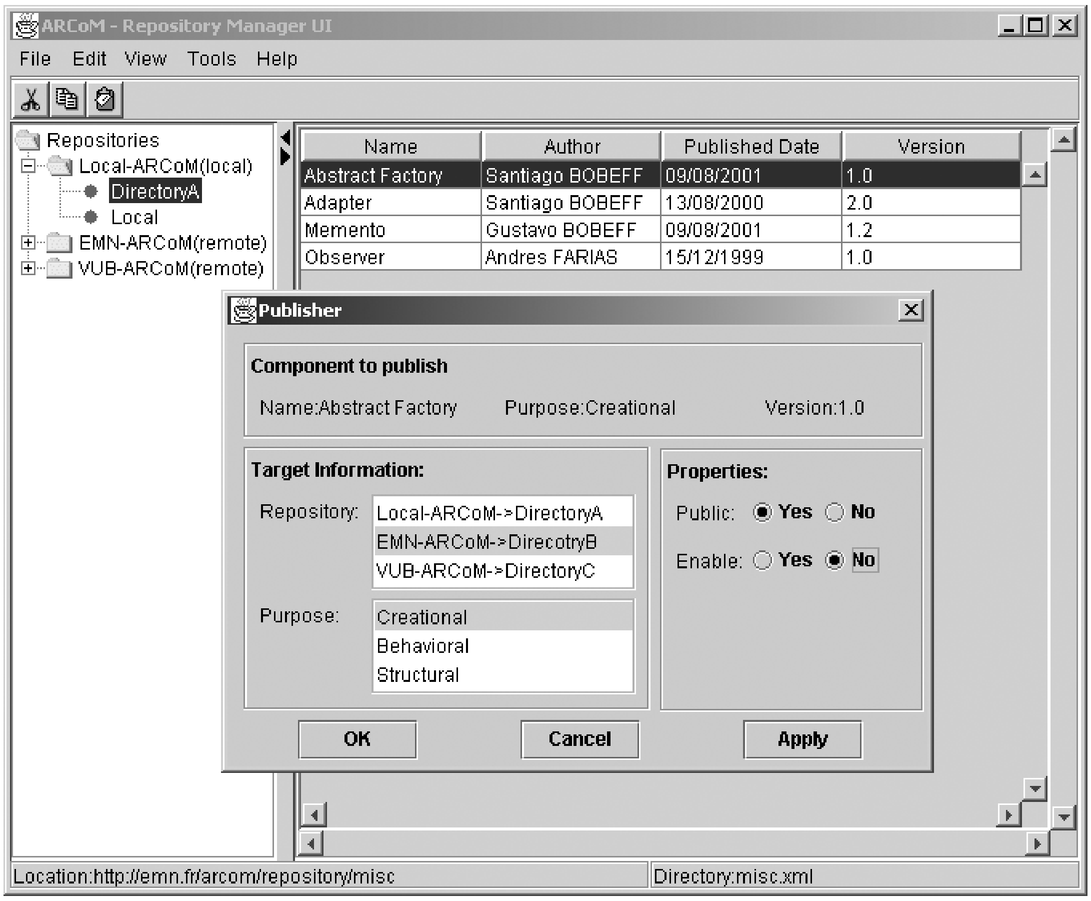
Operations: afterRange

Component: RecurrentEvent

Figure A.3: Constraint Editor's user-interface.

Graph's Element		Adaptation	Comments
Class	General	Rename	Change the class's identifier.
		Modify Structure	Add attributes.
		Modify Behavior	Add operations.
	Attribute	Add	(Similar to Modify Structure)
		Rename	Change the attribute's identifier.
		Modify	Change attribute's type.
		Delete	Delete an attribute.
	Operation	Add	(Similar to Modify Behavior)
		Rename	Change operation's identifier.
		Modify input	Change parameters.
		Modify output	Change output's type.
		Delete	Delete operation.
	Association	Add	It means that the class can be associated.
		Rename	Change association's role.
		Modify multiplicity	Change association's multiplicity.
		Modify end	Change association's type.
	Specialization	Add	Add a new subclass.
		Modify	Change specialization's multiplicity.

Table A.1: Adaptations in ARCoM components.



Repositories
- Local-ARCoM(local)
  - DirectoryA
  - Local
- EMN-ARCoM(remote)
- VUB-ARCoM(remote)

Name	Author	Published Date	Version
Abstract Factory	Santiago BOBEFF	09/08/2001	1.0
Adapter	Santiago BOBEFF	13/08/2000	2.0
Memento	Gustavo BOBEFF	09/08/2001	1.2
Observer	Andres FARIAS	15/12/1999	1.0

Publisher

Component to publish

Name:Abstract Factory Purpose:Creational Version:1.0

Target Information:

Repository: Local-ARCoM->DirectoryA / EMN-ARCoM->DirecotryB / VUB-ARCoM->DirectoryC

Purpose: Creational / Behavioral / Structural

Properties:

Public: [✓ Yes] No

Enable: Yes [✓ No]

Location:http://emn.fr/arcom/repository/misc Directory:misc.xml

Figure A.4: Repository Manager's user-interface.

- To constraint the reuse of a component when the reuse rule states this limitation.

- To generates statistic information about the reuse of component.

A.2.5 Consumer tool

The reuse of an ARCoM component can be achieved by using the ARCoM-Consumer tool. The *consumer* uses this application to select a component from the repositories and then he might decide to publish it by using the ARCoM-Producer tool (see section A.2.1). Figure A.5 shows the implementation of a user interface for that tool.

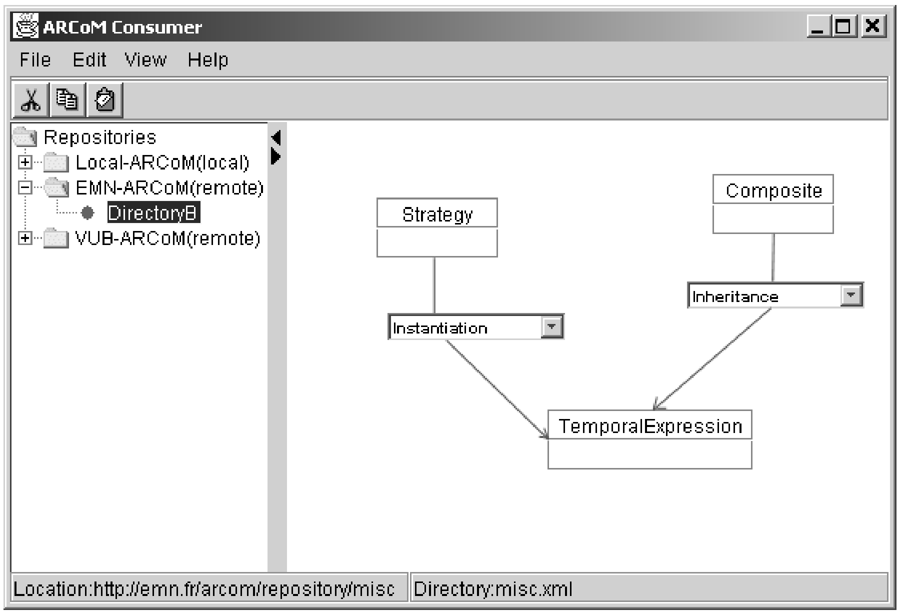


Figure A.5: ARCoM consumer's user-interface.

A.2.6 Software Libraries

The implementation's design provides for packages that permit to manipulate ARCoM components without depending on the user interface applications described above.

Existing Packages

Some services that have been provided by some packages use some existing applications, namely:

- **Tigris-ArgoUML**[4] **v0.8.1a**: We used this tool to provide a graphical interface to the user to generate the UML diagram considered part of the ARCoM model.
- **Apache-Xerces**[5] **v1.4.2**: This general purpose XML parser is used:
  - To manipulate the XML files involved in the application (see Appendix B), for example, generate, and modify the XML generated, check whether a file is ARCoM-compliant or not, etc.

[4]http://argouml.tigris.org/
[5]http://xml.apache.org

Name	Comments	Some classes
arcom.component	Includes the classes to represent the component's representation.	Component, Fundamentals, Specifics, Reuserule, ARCoMBeanInfo, etc.
arcom.repository	Includes the classes that implement the repository's manipulation.	Repository, Directory, Server, RepositoryUI(see figure A.4), etc..
arcom.invariants	Includes the classes to manipulate invariants.	Invariant, InvariantUI (see figure A.3))
arcom.producer	Includes the classes implementing the producer's user interface shown in figure A.1.	ProducerUI.
arcom.consumer	Includes the classes implementing the consumer's user interface shown in figure A.5 .	ConsumerUI.

Table A.2: Packages implemented exclusively for ARCoM's prototype.

  – To process the XMI files, that is class diagram obtained from the UML editor aforementioned, in order to calculate the attributes specified by means a class diagram (see section A.2.2).

- **IBM-OCL Parser[6] v0.3**: This application was integrated to the prototype to manipulate OCL expressions involved in the ARCoM component's description.

Implemented Packages

Table A.2 enumerate the packages implemented to manipulate ARCoM components exclusively.

Table A.3 shows part of the ARCoM bean implementation. This permits to use a ARCoM component in any IDE.

[6]http://www-4.ibm.com/software/ad/library/standards/ocl.html

```
public class ARCoMBeanInfo extends SimpleBeanInfo {
...
   public PropertyDescriptor[] getPropertyDescriptors() {
      try {
         PropertyDescriptor fundamentals =
         new PropertyDescriptor("fundamentals", FundamentalsBean);
         PropertyDescriptor specifics =
         new PropertyDescriptor("specifics", SpecificsBean);
         PropertyDescriptor reuserules =
         new PropertyDescriptor("reuserules", ReuserulesBean);
         PropertyDescriptor label =
         new PropertyDescriptor("label", beanClass);
         fundamentals.setBound(true);
         specifics.setBound(true);
         reuserules.setBound(true);
         ...
         label.setBound(true);
         PropertyDescriptor rv[] =
         {fundamentals, specifics, reuserules,... label};
         return rv;
         } catch (IntrospectionException e) {
         throw new Error(e.toString());
         }
      }}
}
```

Table A.3: ARCoM's bean information class.

A.2.7 System Requirements

To use *ARCoM-Producer* or *ARCoM-Consumer* applications JDK 1.2 or later is required. For *ARCoM-Repository Manager* as server mode was needed to have installed an http server.

Appendix B

XML files

This section includes a description, in details, of the XML files generated to support the administration of the ARCoM components. They are the XML schema files [W3C00, W3C01] that describe the data structure representing the information about the components. Although the model only use two schemas, we will explain an auxiliary one used by the prototype to preserve the information about the repositories, a repository's bookmark file.

B.1 XML schema file for the ARCoM components

To achieve a better undertanding on the structure of this file its content is divided in three parts: *header*, *body* and *user's datatypes* parts.

B.1.1 Header part

The information sections comprised in this part of the file are:

- General data about the component (e.g. creational date, etc.).
- Indexing data to allows the right location of the component in the eventual repository assigned.
- Subscription data that enumerates the many components that support the current component definition (see section B.1.3).

- Publication data that permits the consumer application to check the component's availability to be reused.
- Author data that provides owner's references.

Table B.1 shows the complete definition of this part.

B.1.2 Body part

The content of this part obeys mainly to the structure defined in section 4.1. Additionally to the fundamentals, specifics, reuse rules parts, we include an implementation-oriented part to preserve the *adaptations* made on the original components after a reuse process. Table B.2 shows the complete definition of this part.

B.1.3 User's datatypes part

If we pay attention to the previous definitions (section B.1 and B.2) we find type references that are user defined ones. A detailed definition of them is depicted in this section.

ARCoMNameSpaceType

In table B.1, the expression *<xs:schema xmlns:xs = 'http://www.w3.org/2001/ XMLSchema'>* means that the prefixed (*xs*) data types are associated to this *namespace*, which is the defualt one for the XML files[W3C01]. The same concept is applied for the ARCoM's *repositories*, the place where a component can be published to be reused. To associate the different graph's elements to the right component is used a namespace reference, in this case an ARCoM component is a *namespace*. It makes sense when the element symbolizes a reference of an element belonging to another component. If it is not a reference, that is a copy or an original element, the namespace points to the current component. Table B.3 shows the *ARCoMNameSpacetype* definition included in the schema file.

GraphType

This type makes reference to three user-defined types in order to build the graph idea stated by the model, that is, ClassifierType (see table B.5) to represent the

```
<!—- BEGIN HEADER PART —->
<!– Indexing and general data –>
<xs:attribute name=”name” type=”xs:string”/>
<xs:attribute name=”purpose” type=”xs:string”/>
<xs:attribute name=”prefix” type=”xs:string”/>
<xs:attribute name=”creationDate” type=”xs:date”/>
<xs:attribute name=”schemaVersion” type=”xs:string”/>
<xs:sequence>
 <!– Publication data –>
 <xs:element name=”publication” minOccurs=”0” maxOccurs=”unbounded”>
  <xs:complexType>
   <xs:attribute name=”id” type=”xs:integer”/>
   <xs:attribute name=”lastModificationDate” type=”xs:date”/>
   <xs:sequence>
    <xs:element name=”nameSpace” type=”ARCoMNameSpaceType”/>
   </xs:sequence>
  </xs:complexType>
 </xs:element>
 <!– Subscription data –>
 <xs:element name=”subscription” minOccurs=”0” maxOccurs=”unbounded”>
  <xs:complexType>
   <xs:attribute name=”prefix” type=”xs:string”/>
   <xs:sequence>
    <xs:element name=”nameSpace” type=”ARCoMNameSpaceType”/>
   </xs:sequence>
  </xs:complexType>
 </xs:element>
 <!– Author data –>
 <xs:element name=”author”>
  <xs:complexType>
   <xs:attribute name=”name” type=”xs:string”/>
   <xs:attribute name=”email” type=”xs:uriReference”/>
  </xs:complexType>
 </xs:element>
<!—- END HEADER PART —->
```

Table B.1: XML schema file (header).

```
 <!—- BEGIN BODY PART —->
  <!– Fundamentals part –>
  <xs:element name="fundamentals">
   <xs:complexType>
    <xs:sequence>
     <xs:element name="graph" type="GraphType"/>
     <xs:element name="contextClass" type="IdentifierType"
      minOccurs="0" maxOccurs="unbounded"/>
     <xs:element name="dominantClass" type="IdentifierType"
      minOccurs="0" maxOccurs="unbounded"/>
     <xs:element name="invariants" type="InvariantType"
      minOccurs="0"/>
    </xs:sequence>
   </xs:complexType>
  </xs:element>
  <!– Specifics part –>
  <xs:element name="specifics">
   <xs:complexType>
    <xs:attribute name="applicability" type="xs:string"/>
    <xs:sequence>
     <xs:element name="umlDiagram" type="UMLDiagramType"
      minOccurs="1" maxOccurs="unbounded"/>
     <xs:element name="codeGenerator" type="CodeGenerationRuleType"
      minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
   </xs:complexType>
  </xs:element>
  <!– Reuse Rules part –>
  <xs:element name="reuserules">
   <xs:complexType>
    <xs:sequence>
     <xs:element name="rule" type="ReuseRuleType" minOccurs="0"/>
    </xs:sequence>
   </xs:complexType>
  </xs:element>
  <!– Adaptation log –>
  <xs:element name="adaptations">
   <xs:complexType>
    <xs:sequence>
     <xs:element name="adaptation"
      type="AdaptationType" minOccurs="0"/>
    </xs:sequence>
   </xs:complexType>
  </xs:element>
 </xs:sequence>
</xs:complexType>(from header)
 <!—- END BODY PART —->
```

Table B.2: XML schema file (body).

```
<xs:complexType name="ARCoMNameSpaceType">
 <xs:attribute name="location"type="xs:uriReference"/>
 <xs:attribute name="directoryName" type="xs:string"/>
</xs:complexType>
```

Table B.3: ARCoMNameSpaceType type's definition.

```
<xs:complexType name="GraphType">
 <xs:sequence>
  <xs:element name="C">
   <xs:complexType>
    <xs:sequence>
     <xs:element name="classifier" type="ClassifierType"
      minOccurs="1" maxOccurs="unbounded"/>
    </xs:sequence>
   </xs:complexType>
  </xs:element>
  <xs:element name="L">
   <xs:complexType>
    <xs:sequence>
     <xs:element name="generalization" type="GeneralizationType"
      minOccurs="0" maxOccurs="unbounded"/>
     <xs:element name="association" type="AssociationType"
      minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
   </xs:complexType>
  </xs:element>
 </xs:sequence>
</xs:complexType>
```

Table B.4: GraphType type's definition.

C set, and GeneralizationType, AssociationType, RealizationType (see tables B.8, B.10, respectively) to represent the L set. For this version we consider as the possible relationships among classes, inheritance and the different kind of associations. Table B.4 shows the complete definition.

ClassifierType

The attribute structure of this type was, mainly, copied form the *classifier* definition for XMI[Gro99]. It uses *AttributeType* and *OperationType* (shown in tables B.18 and B.7, respectively) to define the class structure. Table B.5 shows the complete definition.

```
<xs:complexType name="ClassifierType">
 <xs:attribute name="name" type="xs:string"/>
 <xs:attribute name="type" default="Class">
  <xs:simpleType>
   <xs:restriction base = "xs:string">
    <xs:enumeration value="Class"/>
    <xs:enumeration value="Interface"/>
    <xs:enumeration value="Datatype"/>
    <xs:enumeration value="Signal"/>
    <xs:enumeration value="Component"/>
    <xs:enumeration value="Node"/>
    <xs:enumeration value="Use case"/>
    <xs:enumeration value="Subsystem"/>
   </xs:restriction>
  </xs:simpleType>
 </xs:attribute>
 <xs:attribute name="visibility" default="public">
  <xs:simpleType>
   <xs:restriction base = "xs:string">
    <xs:enumeration value="public"/>
    <xs:enumeration value="protected"/>
    <xs:enumeration value="private"/>
   </xs:restriction>
  </xs:simpleType>
 </xs:attribute>
 <xs:attribute name="isAbstract" type="xs:boolean"/>
 <xs:attribute name="isRoot" type="xs:boolean"/>
 <xs:attribute name="isLeaf" type="xs:boolean"/>
 <xs:attribute name="isPolymorphic" type="xs:boolean"/>
 <xs:attribute name="multiplicity" type="xs:integer"/>
 <xs:sequence>
  <xs:element name="identifier" type="xs:IdentifierType">
  <xs:element name="features">
   <xs:complexType>
    <xs:sequence>
     <xs:element name="attribute" type="AttributeType"
      minOccurs="0" maxOccurs="unbounded"/>
     <xs:element name="operation" type="OperationType"
      minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
   </xs:complexType>
  </xs:element>
 </xs:sequence>
</xs:complexType>
```

Table B.5: ClassifierType type's definition.

```
<xs:complexType name="AttributeType">
 <xs:attribute name="prefixNameSpace" type="xs:string"/>
 <xs:attribute name="idInNameSpace" type="xs:integer"/>
 <xs:attribute name="name" type="xs:string"/>
 <xs:attribute name="visibility" type="xs:string"/>
 <xs:attribute name="multiplicity" type="xs:string"/>
 <xs:attribute name="type" type="xs:string"/>
 <xs:attribute name="initValue" type="xs:string"/>
 <xs:attribute name="adaptation" type="xs:string"/>
 <xs:attribute name="property" default="none">
  <xs:simpleType>
   <xs:restriction base = "xs:string">
    <xs:enumeration value="none"/>
    <xs:enumeration value="changeable"/>
    <xs:enumeration value="addOnly"/>
    <xs:enumeration value="forzen"/>
   </xs:restriction>
  </xs:simpleType>
 </xs:attribute>
 <xs:sequence>
  <xs:element name="identifier" type="xs:IdentifierType">
 </xs:sequence>
</xs:complexType>
```

Table B.6: AttributeType type's definition.

AttributeType and OperationType

Like *ClassifierType*, their structure was, mainly, copied form the *feature* definition for XMI[Gro99]. They are identified to implement a tracing process for the adaptation generated (see table A.1). Tables B.6 and B.7 show the complete definition, respectively.

GeneralizationType and AssociationType

They express the two possible relationship established among the classifier included in the solution. Like *ClassifierType*, their structure was, mainly, copied form the *Assosiations and Generalizable* definition for XMI[Gro99]. They are identified to implement a tracing process for the adaptation generated (see table A.1). Tables B.8 and B.9 show the complete definition, respectively.

```
<xs:complexType name="OperationType">
 <xs:attribute name="name" type="xs:string"/>
 <xs:attribute name="visibility" type="xs:string" use="optional"/>
 <xs:attribute name="returnType" type="xs:string"/>
 <xs:attribute name="adaptation" type="xs:string"/>
 <xs:attribute name="property" use="optional">
  <xs:simpleType>
   <xs:restriction base = "xs:string">
    <xs:enumeration value="leaf"/>
    <xs:enumeration value="isQuery"/>
    <xs:enumeration value="sequencial"/>
    <xs:enumeration value="guarded"/>
    <xs:enumeration value="concurrent"/>
   </xs:restriction>
  </xs:simpleType>
 </xs:attribute>
 <xs:sequence>
  <xs:element name="identifier" type="xs:IdentifierType">
  <xs:element name="parameter" minOccurs="0" maxOccurs="unbounded">
   <xs:complexType>
    <xs:sequence>
     <xs:element name="identifier" type="xs:IdentifierType">
    </xs:sequence>
    <xs:attribute name="name" type="xs:string"/>
    <xs:attribute name="adaptation" type="xs:string"/>
    <xs:attribute name="direction" use="optional">
     <xs:simpleType>
      <xs:restriction base = "xs:string">
       <xs:enumeration value="in"/>
       <xs:enumeration value="out"/>
       <xs:enumeration value="inout"/>
      </xs:restriction>
     </xs:simpleType>
    </xs:attribute>
    <xs:attribute name="type" type="xs:string"/>
    <xs:attribute name="defaultType" type="xs:string" use="optional"/>
   </xs:complexType>
  </xs:element>
 </xs:sequence>
</xs:complexType>
```

Table B.7: OperationType type's definition.

```
<xs:complexType name="GeneralizationType">
 <xs:attribute name="name" type="xs:string"/>
 <xs:attribute name="adaptation" type="xs:string"/>
 <xs:sequence>
  <xs:element name="identifier" type="xs:IdentifierType">
  <xs:element name="parent" type="IdentifierType"/>
  <xs:element name="child" type="IdentifierType"/>
 </xs:sequence>
</xs:complexType>
```

Table B.8: GeneralizationType type's definition.

```
<xs:complexType name="AssociationType">
 <xs:attribute name="name" type="xs:string"/>
 <xs:sequence>
  <xs:element name="identifier" type="xs:IdentifierType">
  <xs:element name="end" type="EndType" minOccurs="2" maxOccurs="2"/>
 </xs:sequence>
</xs:complexType>
```

Table B.9: AssociationType type's definition.

AssociationEndType

It permits the treatment of the end of every association separately. Like *ClassifierType*, its structure was, mainly, copied form the *AssosiationEnd* definition for XMI[Gro99]. Table B.10 shows the complete definition.

IdentifierType

Since the elements involved in the representation belong to a *nameSpace* (see table B.1.3), we use the *IdentifierType* to implement this addressing system. An identifier is compound by two attributes:

- **prefixNameSpace**: It is a reference to a *nameSpace* listed in the Subscription part (see table B.1). For the element defined in current component this attribute has the same value than that assigned to the *prefix* attribute (see Indexing and General data part in table B.1).
- **idInNameSpace**: It is a identifier of an element in the *nameSpace* specified in the previous attribute.

Table B.11 shows the complete definition.

```
<xs:complexType name="AssociationEndType">
 <xs:attribute name="name" type="xs:string"/>
 <xs:attribute name="isNavegable" type="xs:boolean"/>
 <xs:attribute name="type">
  <xs:simpleType>
   <xs:restriction base = "xs:string">
    <xs:enumeration value="Aggregation"/>
    <xs:enumeration value="Composition"/>
    <xs:enumeration value="none"/>
   </xs:restriction>
  </xs:simpleType>
 </xs:attribute>
 <xs:sequence>
  <xs:element name="multiplicity" type="MultiplicityType"/>
  <xs:element name="associated" type="IdentifierType"/>
 </xs:sequence>
</xs:complexType>
```

Table B.10: AssociationEndType type's definition.

```
<xs:complexType name="IdentifierType">
 <xs:attribute name="prefixNameSpace" type="xs:string"/>
 <xs:attribute name="idInNameSpace" type="xs:integer"/>
</xs:complexType>
```

Table B.11: IdentifierType type's definition.

```
<xs:complexType name="MultiplicityType">
 <xs:attribute name="lower" type="xs:nonNegativeInteger"/>
 <xs:attribute name="upper" type="xs:string"/>
</xs:complexType>
```

Table B.12: MultiplicityType type's definition.

```
<xs:complexType name="InvariantType">
 <xs:sequence>
  <xs:element name="identifier" type="IdentifierType"/>
  <xs:element name="expresion" type="xs:string"/>
 </xs:sequence>
</xs:complexType>
```

Table B.13: InvariantType type's definition.

MultiplicityType

It is used by *ClassifierType* and *AssociationEndType* to specify their multiplicity. Table B.12 shows the complete definition.

InvariantType

The attributes that imply invariants specifications, that is, *Adaptation Invariants* and *Structural Invariants*, use this definition. Table B.13 shows the complete definition.

FileType

The information about the files related to the component representation is determined by this type. It is used for the UML diagrams generated by the UML Editor. Table B.14 shows the complete definition.

```
<xs:complexType name="FileType">
 <xs:attribute name="name" type="xs:string"/>
 <xs:attribute name="location" type="xs:string"/>
</xs:complexType>
```

Table B.14: FileType type's definition.

```
<xs:complexType name=”UMLDiagramType”>
 <xs:attribute name=”name” type=”xs:string”/>
 <xs:attribute name=”type”>
  <xs:simpleType>
   <xs:restriction base = ”xs:string”>
    <xs:enumeration value=”Class”/>
    <xs:enumeration value=”Sequence”/>
    <xs:enumeration value=”Usa Case”/>
    <xs:enumeration value=”State”/>
    <xs:enumeration value=”Activity”/>
    <xs:enumeration value=”Deployment”/>
   </xs:restriction>
  </xs:simpleType>
 </xs:attribute>
 <xs:sequence>
  <xs:element name=”file” type=”FileType”/>
 </xs:sequence>
</xs:complexType>
```

Table B.15: UMLDiagramType type’s definition.

UMLDiagramType

Although this version of the model only takes into account two UML diagram, Class and Sequence, as part of the *specifics*, we define this generic type to allow futures extensions. Table B.14 shows the complete definition.

ReuseRuleType

Defines the *reuse rule’s* structure. Table B.16 shows the complete definition.

CodeGenerationRuleType

To specify the rules for code generation we define this generic type. In order to improve this facility, a more elaborated analysis should be done to redefine this element. Table B.17 shows the complete definition.

AdaptationType

This type was defined to log the eventual adaptations on the components. Table B.18 shows the complete definition.

```
<xs:complexType name="ReuseRuleType">
 <xs:attribute name="name" type="xs:string"/>
 <xs:attribute name="dependence" type="xs:boolean"/>
 <xs:attribute name="predominance" type="xs:boolean"/>
 <xs:attribute name="shareability" type="xs:string"/>
 <xs:attribute name="transitiveness" type="xs:boolean"/>
 <xs:attribute name="method"/>
  <xs:simpleType>
   <xs:restriction base = "xs:string">
    <xs:enumeration value="Copy"/>
    <xs:enumeration value="Reference"/>
   </xs:restriction>
  </xs:simpleType>
 </xs:attribute>
 <xs:sequence>
  <xs:element name="identifier" type="xs:IdentifierType">
  <xs:element name="propagation">
   <xs:complexType>
    <xs:attribute name="range"/>
     <xs:simpleType>
      <xs:restriction base = "xs:string">
       <xs:enumeration value="Complete"/>
       <xs:enumeration value="Partial"/>
      </xs:restriction>
     </xs:simpleType>
    </xs:attribute>
    <xs:sequence>
     <xs:element name="graphElementFlood" type="IdentifierType"
      minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
   </xs:complexType>
  </xs:element>
  <xs:element name="connector">
   <xs:complexType>
    <xs:attribute name="relationship" type="xs:string"/>
    <xs:sequence>
     <xs:element name="graphElement" type="IdentifierType"/>
    </xs:sequence>
   </xs:complexType>
  </xs:element>
  <xs:element name="adaptationConstraint" type="InvariantType"/>
 </xs:sequence>
</xs:complexType>
```

Table B.16: ReuseRuleType type's definition.

```
<xs:complexType name="CodeGenerationRuleType">
 <xs:attribute name="language" type="xs:string"/>
 <xs:sequence>
  <xs:element name="expression" type="xs:string"/>
 </xs:sequence>
</xs:complexType>
```

Table B.17: CodeGenerationRuleType type's definition.

```
<xs:complexType name="AdaptationType">
 <xs:attribute name="operation">
  <xs:simpleType>
   <xs:restriction base = "xs:string">
    <xs:enumeration value="Add"/>
    <xs:enumeration value="Rename"/>
    <xs:enumeration value="Modify"/>
    <xs:enumeration value="Delete"/>
   </xs:restriction>
  </xs:simpleType>
 </xs:attribute>
 <xs:attribute name="value" type="xs:string"/>
 <xs:complexType>
  <xs:sequence>
   <xs:element name="identifier" type="IdentifierType"/>
  </xs:sequence>
 </xs:complexType>
</xs:complexType>
```

Table B.18: AdaptationType type's definition.

```
<?xmlversion="1.0"encoding="UTF-8"?>
 <xs:schema xmlns:xs='http://www.w3.org/2001/XMLSchema'>
 <arcom:schema xmlns:xs='arcom_component.xsd'>
 <xs:element name="component" type="ARCoMRepositoryType"
  minOccurs="0" maxOccurs="unbounded"/>
 <!– ARCoM Component Entry –>
 <xs:complexType name="ARCoMRepositoryType">
   <xs:attribute name="id" type="xs:integer"/>
   <xs:attribute name="name" type="xs:string"/>
   <xs:attribute name="purpose" type="xs:string"/>
   <xs:attribute name="isPublic" type="xs:boolean"/>
   <xs:attribute name="isEnable" type="xs:boolean"/>
   <xs:attribute name="publishedDate" type="xs:date"/>
   <xs:sequence>
     <xs:element name="fileVersion" type="arcom:FileType">
     <xs:element name="versionList">
       <xs:complexType>
         <xs:elememnt name="id" type="xs:integer" minOccurs="0"
          maxOccurs="unbounded"/>
       </xs:complexType>
     </xs:element>
   </xs:sequence>
 </xs:complexType>
</xs:schema>
```

Table B.19: Repository's XML schema file.

B.2 Repository schema

ARCoMNameSpace type provides the needed structure to represent a directory address. They can be located by specifying a *location*, which is a machine address plus a protocol, a directory, which is place in the machine, and finally a filename which contents the information about the published components. Despite the extra effort that a distribute management implies, the distributed manipulation of the component references permits spread the component solution beyond its building environment. Table B.19 shows the XML schema file representing the data structure of the repositories.

B.3 Bookmark schema

This schema file is used by the *Repository Manager* (see section A.2.4) to conserve a reference to the repository's location. It is a bookmark file. Table B.20 shows the complete definition.

```
<?xml version="1.0" encoding="UTF-8"?>
 <xs:schema xmlns:xs='http://www.w3.org/2001/XMLSchema'>
 <arcom:schema xmlns:xs='arcom_component.xsd'>
 <xs:element name="repository" type="ARCoMBookmarkType"
  minOccurs="0" maxOccurs="unbounded"/>
 <!– ARCoM Repository Entry –>
 <xs:complexType name="ARCoMRepositoryType">
  <xs:attribute name="alias" type="xs:integer"/>
  <xs:attribute name="type" type="xs:string"/>
  <xs:attribute name="lastAccessDateTime" type="xs:timePeriod"/>
  <xs:attribute name="passRequired" type="xs:boolean"/>
  <xs:sequence>
   <xs:element name="location" type="arcom:ARCoMNameSpaceType">
   </xs:element>
  </xs:sequence>
 </xs:complexType>
</xs:schema>
```

Table B.20: Bookmark's XML schema file.

Appendix C

Concrete Example

In this section we develop a solution for a concrete problem by using ARCoM's specifications.

C.1 Problem

Martin Fowler[Fow98] describes a problem, called '*Recurring Events for Calendars*'. This problem implies, in few words, the necessity to manipulate expressions that represent recurrent events in a calendar. The requirements consider the *temporal expressions* listed in table C.1. The possible combinations of *temporal expressions* include set operations, like *intersection*, *union* and *difference.* The resulting set of recurrent events from the mentioned operations should be considered also a *temporal expression.*

Additionally the problem's description requires to be able to make some queries to the calendar:

- Given an *event* and a *date range* would be useful to know how many *occurrences* of *event* appear during the *date range.*
- Another should be to get the next *occurrence* of an *event* given an initial *date.*
- And finally to determine whether an *event* would occur on a given *date*

According to these requirements we will describe a suggested solution in terms of ARCoM components.

Name	Problem	Solution
Day Every Month	You need to represent statements of the form 2nd Monday of the Month.	Use a day every month temporal expression with a day of the week and a count.
Range Every Year	You need to represent statements of the form 14 March till 12 October	Use a range every year temporal expression with a day and month at the start and end.
Set Expression	You need to represent combinations of temporal expressions.	Define set combinations for union, intersection and difference.

Table C.1: Temporal expressions.

C.2 Solution

Let see a resumed description oriented to fit with a well-known design pattern. Fowler[Fow98] describes a problem where we need to manipulate similar abstractions in different ways depending on the internal structure of every abstraction. The set of queries that every abstraction has to be able to answer is the same for the whole set of *temporal expression*, in this case. Each *temporal expression* implements its own strategy to solve those queries. Concretely we will use *strategy pattern* to solve this problem. Hence we pick an ARCoM component that expresses this design solution. Figure C.1 illustrates the original pattern that is being reused by using the *instantiation* rule for the *TemporalExpression* component. Table C.2 shows the pattern XML file where are defined the reuse rules.

TemporalExpression component needs to be adapted to fit the requirements of the particular problem. Figure C.2 shows the class diagram associated to the component, after adaptations. These alterations will not be restricted because none Invariant and Adaptation Constraint was defined.

Following the procedure to become the resulting component in reusable one, the corresponding *reuse rules* have to be specified. Table C.3 shows the reuse rule defined for *TemporalExpression*, therefore it can be now reused by any component.

So far we have built a component that meets part of the requirements in the Fowler's problems. We defined only the simpler *temporal expression*, but to provide a complete solution it is necessary to include the modelling of the combined *temporal expression*, that is, *Intersection* and *Union*.

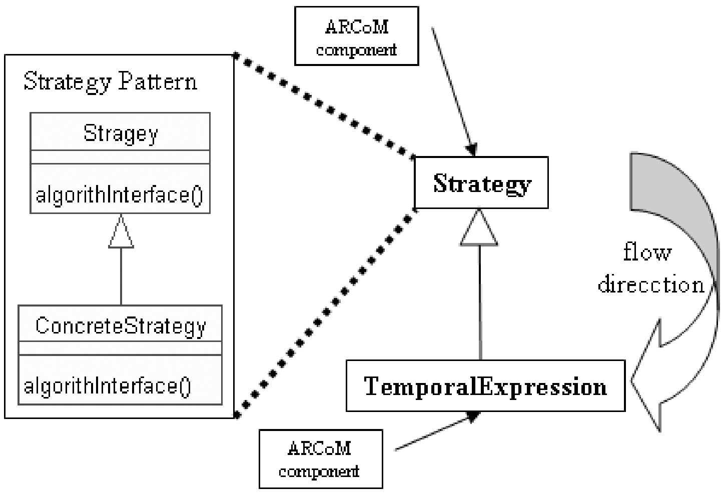


Figure C.1: TemporalExpression component by the instantiation of the Strategy component.

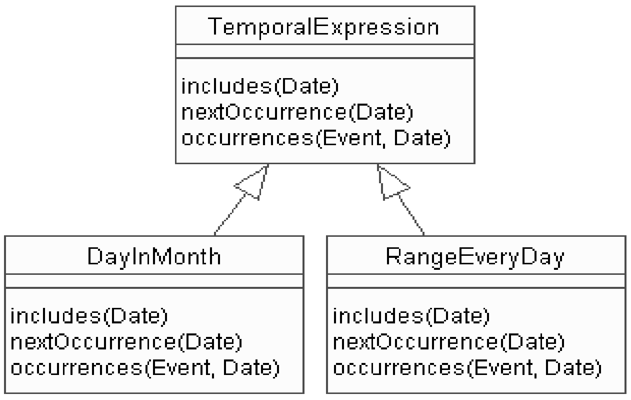


Figure C.2: Adapting the TemporalExpression component.

```
<?xml version="1.0" encoding="UTF-8"?>
 <component xmlns:xs='http://www.w3.org/2001/XMLSchema'
 xs:noNamespacesSchemaLocation='personal.xsd' name="Strategy"
 purpose="Behavioral" prefix ="str" creationDate ="2001/07/25" schemaVersion ="0.1">
  <publication id ="1" lastModificationDate ="2001/08/15">
   <nameSpace location ="http://emn.fr/arcom" directoryName ="gangoffour.xml"/>
  </publication>
  <!– Subscription data empty –>
  <author name ="Gustavo BOBEFF" email ="gbobeff@emn.fr"/>
  <fundamentals>
   <graph>
    <C>
    <classifier name="Strategy" type="Class" visibility="public"
    isAbstract="true" isRoot="false" isLeaf="false" isPolymorphic="true">
     <identifier prefixNameSpace="str" idInNameSpace="1"/>
      <features>
       <operation name="AlgorithInterface" visibility="public" returnType="void"
        <identifier prefixNameSpace="str" idInNameSpace="2"/>
       </operation>
      </features>
    </classifier>
    <classifier name="ConcreteStrategy" type="Class" visibility="public"
    isAbstract="false" isRoot="false" isLeaf="false" isPolymorphic="true">
     <identifier prefixNameSpace="str" idInNameSpace="3"/>
    </classifier>
    </C>
    <L>
    <generalization name="xs:string" adaptation="xs:string">
     <identifier prefixNameSpace="str" idInNameSpace="5"/>
     <parent prefixNameSpace="str" idInNameSpace="1"/>
     <child prefixNameSpace="str" idInNameSpace="3"/>
    </generalization>
    </L>
   </graph>
   <contextClass prefixNameSpace="str" idInNameSpace="1"/>
   <dominantClass prefixNameSpace="str" idInNameSpace="1"/>
  </fundamentals>
  <specifics applicability="xs:string">
   <umlDiagram name="xs:string" type="xs:string">
    <file name="xs:string" location="xs:string"/>
   </umlDiagram>
   <codeGenerator language="xs:string">
    <rule>
     <value></value>
    </rule>
   </codeGenerator>
  </specifics>
  <reuserules>
   <rule name="inheritance" dependence="true" predominance="false"
   shareability="true" transitiveness="true" method="reference">
    <identifier prefixNameSpace="str" idInNameSpace="6"/>
    <propagation range="complete"></propagation>
    <connector relationship="xs:string">
     <graphElement prefixNameSpace="str" idInNameSpace="1"/>
    </connector>
    <adaptationInvariants><adaptationInvariants>
   </rule>
   <rule name="instantiation" dependence="false" predominance="false"
   shareability="true" transitiveness="true" method="copy">
    <identifier prefixNameSpace="str" idInNameSpace="7"/>
    <propagation range="complete"></propagation>
    <adaptationInvariants></adaptationInvariants>
   </rule>
  </reuserules>
</component>
```

Table C.2: Strategy in an ARCoM's XML file.

```
<reuserules>
 ...
 <rule name=”inheritance” dependence=”true” predominance=”false”
 shareability=”true” transitiveness=”true” method=”reference”>
  <identifier prefixNameSpace=”str” idInNameSpace=”7”/>
  <propagation range=”complete”></propagation>
  <connector relationship=”inheritance”>
   <graphElement prefixNameSpace=”str” idInNameSpace=”1”>
  </connector>
  <adaptationInvariants></adaptationInvariants>
 </rule>
 ...
 <rule name=”composition” dependence=”true” predominance=”false”
 shareability=”true” transitiveness=”true” method=”reference”>
  <identifier prefixNameSpace=”str” idInNameSpace=”8”/>
  <propagation range=”complete”></propagation>
  <connector relationship=”composition”>
   <graphElement prefixNameSpace=”str” idInNameSpace=”1”>
  </connector>
  <adaptationInvariants></adaptationInvariants>
 </rule>
</reuserules>
```

Table C.3: Reuse rules in TemporalExpression.

Based on the solution suggested by Fowler[Fow98], those last cases obey to the same solution developed for *DayInMonth* and *RangeEveryDay*. They are treated as collection of temporal expression that implies its own strategies to answer the questions. Figure C.3 shows the resulting component.

But that is not the end. We need to satisfy two requirements. Firstly the fact that the type of the elements in the *collection* is *temporal expression*. Secondly, a *collection* is a *temporal expression*. A solution can be achieved if the *CollectioTE* component reuse *TemporalExpression* component twice, one by inheritance and the second by association(composition), respectively. Figure C.4 illustrate the solution for the recurrent event problem, in ARCoM composition notation.

The internal structure is illustrated by the class diagram in figure C.5.

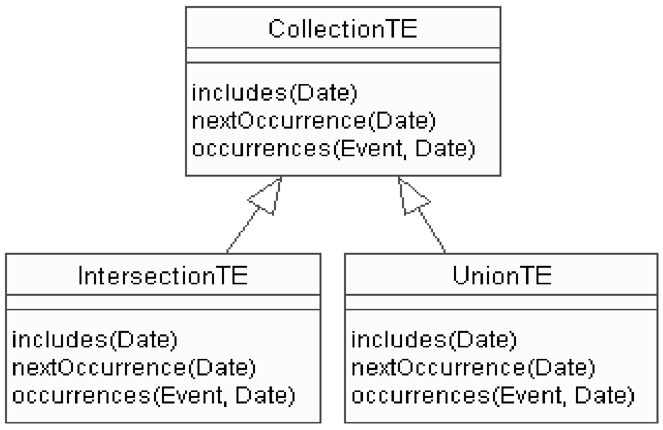


Figure C.3: Collections of temporal expression.

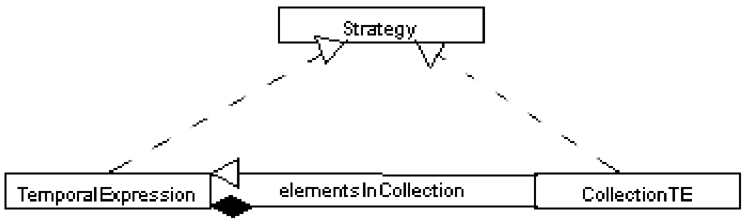


Figure C.4: Solution appearance in the ARCoM Consumer tool.

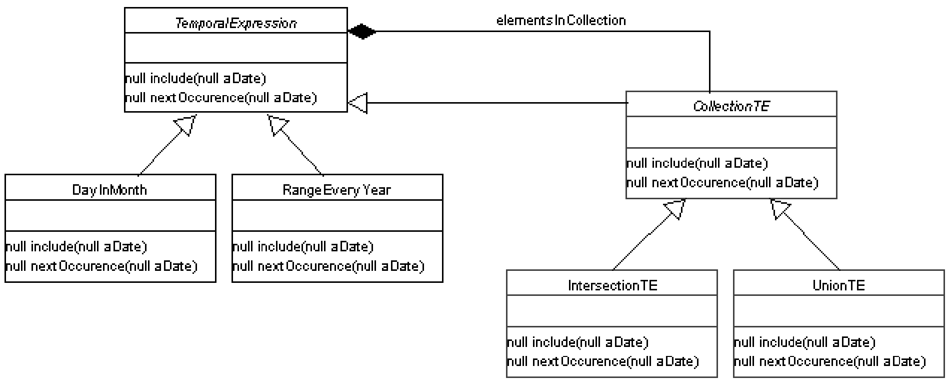


Figure C.5: Final solution's class diagram.

Bibliography

[ABvdSB94] Mehmet Aksit, Jan Bosch, William van der Sterren, and Lodewijk Bergmans. Real-time specification inheritance anomalies and real-time filters. In M. Tokoro and R. Pareschi, editors, *Proceedings ECOOP'94*, pages 386–407, Bologna, Italy, 1994. Springer-Verlag.

[ASC98] Anastasia Analyti, Nicolas Spyratos, and Panos Constantopoulos. On the semantics of a semantic network. *Fundamenta Informaticae*, 36(2-3):109–144, 1998.

[Boo87] G. Booch. Software components with ada, 1987.

[BPM$^+$98] V. Benjamins, E. Plaza, E. Motta, D. Fensel, R. Studer, B. Wielinga, G. Schreiber, Z. Zdrahal, and S. Decker. Ibrow3: An intelligent brokering service for knowledge-component reuse on the world-wide web, 1998.

[BRJ98] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modelling Language User Guide.* Object Technology Series. Addison-Wesley, 1998.

[CD00] John Cheesman and John Daniels. *UML Components. A Simple Process for Specifying Components-Based Software.* Addison-Wesley, 2000.

[CJ94] B. Christine and Marciniak John J. Encyclopedia of software engineering, 1994.

[DeS97] Alden DeSoto. Using the beans development kit 1.0. Tutorial, September 1997.

[Fow98] Martin Fowler. Recurring events for calendars. Technical report, February 1998.

[Fre83] P. Freeman. Reusable software engineering: Concepts and research directions, 1983.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, Massachusetts, 1994.

[Gro97] M. Group. Meta group homepage, 1997.

[Gro99] Object Managment Group. Xmi specification. Technical report, February 1999.

[Hem93] T. Hemmann. Reuse approaches in software engineering and knowledge engineering: A comparison, 1993.

[HL95] Walter Hürsch and Cristina Videira Lopes. Separation of concerns. Technical Report NU-CCS-95-03, Boston, Massachusetts, 24 1995.

[JCJO92] Ivar Jacobsen, Magnus Christerson, Patrik Jonsson, and G. G. Overgaard. *Object-Oriented Software Engineering.* Addison-Wesley, 1992.

[JGJ97] I. Jacobson, M. Griss, and P. Jonsson. Software reuse: Architecture, process and organization for business success, 1997.

[Joh91] L. Johnson. Harris: Sharing and reuse of requirements knowledge; proc, 1991.

[KBV00] M. Sihling K. Bergner, A. Rausch and A. Vilbig. Putting the parts together - concepts, description techniques, and development process for componentware, 2000.

[Kru92] Charles W. Krueger. Software reuse. *ACM Computing Surveys*, 24(2):131–183, June 1992.

[KS98] Rudolf K.Keller and Reinhard Schauer. Design components : Towards software composition at the design level. In *International Conference on Software Engineering*, 1998.

[Kum00] S. Illango Kumaran. Xmi: The specification for component's metadata interchange. Technical report, May 2000.

[Mey94] Bertrand Meyer. *Reusable Software: The Base Object-Oriented Component Libraries.* Prentice Hall, Englewood Cliffs, 1994.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall Professional Technical Reference, 1997.

[Mey99] Bertrand Meyer. Rules for component builders. Technical report, 1999.

[ND95] Oscar Nierstrasz and Laurent Dami. Component-oriented software technology. In Oscar Nierstrasz and Dennis Tsichritzis, editors, *Object-Oriented Software Composition*, pages 3–28. Prentice Hall, 1995.

[Pet91] A. Peterson. Coming to terms with software reuse terminology: a model-based approach, 1991.

[Rui00] Diego Sevilla Ruiz. Corba & components. Tutorial, Noviembre 2000.

[Sam97] Johannes Sametinger. *Software Engineering with Reusable Components*. Springer-Verlag, 1997.

[SN99] J. Schneider and O. Nierstrasz. Scripts and glue, 1999.

[Szy98] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, N.Y., 1998.

[Tra95] Will Tracz. *Confessions of a Used-Program Salesman: Institutionalizing Software Reuse*. Addison Wesley, Reading, MA, 1995.

[W3C00] World Wide Web Consortium W3C. Extensible markup language (xml)1.0 (second edition). Recommendations, February 2000.

[W3C01] World Wide Web Consortium W3C. Xml schema part 0: Primer. Recommendations, May 2001.

[WK98] J. Warmer and A. Kleppe. *The Object Constraint Language*. Object Technology Series. Addison-Wesley, 1998.