Design and Implementation of a Reflective Component-Oriented Programming and Modeling Language

by

Petr SPACEK

ACADÉMIE DE MONTPELLIER

UNIVERSITÉ MONTPELLIER II

— SCIENCES ET TECHNIQUES DU LANGUEDOC —

THÈSE

présentée au Laboratoire d'Informatique de Robotique
et de Microélectronique de Montpellier pour
obtenir le diplôme de doctorat

SPÉCIALITÉ	:	**INFORMATIQUE**
Formation Doctorale	:	**Informatique**
École Doctorale	:	**Information, Structures, Systèmes**

Design and Implementation of a Reflective Component-Oriented Programming and Modeling Language

par

Petr SPACEK

Soutenue le 17 decembre 2013 at ??h, devant le jury composé de :

Lionel SEINTURIER, Professeur, Inria, University Lille 1, France ..... Rapporteur
Ivica CRNKOVIC, Professeur, IDT, Mälardalen University, Sweden, ... Rapporteur
Pierre COINTE, Professeur, LINA, Université de Nantes, France ..... Examinateur
Roland DUCOURNAU, Professeur, LIRMM, Université Montpellier II, France ..... Président
Christophe DONY, Professeur, LIRMM, Université Montpellier II, France ..... Directeur de Thèse
Chouki TIBERMACINE, Associer Professeur, LIRMM, Université Montpellier II, France .. Co-Directeur de Thèse

Contents

Acknowledgement

Saying thank you is more than good manners. It is good spirituality.

Alfred PAINTER.

It hardly seems possible to me that I have finally reached the end of this long process, and that it only remains to thank the people who have helped me along the way. Even beyond the formal aspects of my education, I have learned much and gained much from the people around me over the last seven years, and I know that I will never be able to truly express my appreciation. I can only say: Thank you!

To Christophe and Chouki, I say: Thank you! You have taught me about research, about speaking, and writing, about software architecture, and about life. You have supported me at every stage, from my first arrival here at LIRMM to the final completion of my dissertation. You have supported me when I was coasting along not getting much accomplished and when I was pushing hard to do more than seemed possible. You have helped make my dissertation the best that I can make it. You have been patient with me when I had nothing to give you and responsive when I gave you too much all at once. You have truly gone beyond the call of duty.

To the MAREL group, I say: Thank you! You put up with my half-baked ideas and my buggy prototypes and turned them into something real and even useful. Your feedback and your example have enriched my experience as well as my work.

To my office mates, past and present, I say: Thank you! You have tolerated my muttering and teasing and aggressive work-avoidance tactics. You have helped me to know that it can be done, and to remember that there is more to computer science than my little corner of it.

To my family, and especially to HER (she knows), I say: Thank you! Your support has kept me going when I thought I couldn't go on. You have kept me in perspective, and helped me keep my priorities straight. You are what makes it all worthwhile.

Abstract

Component-based Software Engineering (CBSE), to produce software by connecting of the shelf ready-to-use components, promises costs reduction during the development, the maintenance and the evolution of a software. The recent period has seen the production of a very important set of new results in this field. As the term "*component*" is very general, it encompasses many researches having different objectives and offering various kind of abstractions and mechanisms. However one main overall accepted idea is to model software with components organized into architectures and to generate code from such abstract descriptions. This is a good idea but the question arise to know which languages are good candidate for the generated code. In the current practice the design phase happens in the component world and the programming phase occurs in the object-oriented world. It appears that languages and technologies used to achieve component-based development are only partially component-based. Our first claim is that to use component-based languages to write the executable code is primarily important just because the original component-based designs (*e.g.* requirements, architectures) do not vanish at run-time, making programs more understandable and reversible. By doing this, it is then possible to imagine that design (modeling) and programming can be done at the same conceptual level and why not using the same language. Usually, objects are most always chosen to implements component-based designs. It is true that an object is certainly the existing executable thing the closest to a component as they are understood today; close but not exactly the same. Our second claim is then that it is possible to achieve component-programming languages by smoothly modifying object-oriented ones. Following these ideas, we present in this thesis an example of a new pure component-based programming and modeling language, named COMPO incorporating, in a simple and uniform way, core concepts and mechanisms necessary for the description and implementation of components and of component-based architectures: component, port, service, connection and the following mechanisms: instantiation, service invocation, composition and substitution. We also claim that describing components, their architectures (structures) and their services (behavior) would benefit (as objects descriptions do) from an inheritance-based differential description. In consequence we propose a specification and implementation of an inheritance system taking requirements into account on a covariant specialization policy base and with a corresponding dedicated substitution mechanism. We finally claim that making such a language fully reflective will open an interesting new alternative (in the component's context) for any king of model or program checking or transformation. We revisit some standard solutions to achieve an original component-oriented reification of concepts to build up an executable meta-model designed on the idea of "*everything is a component*". A complete prototype implementation of the COMPO language has been achieved and is described in this thesis.

Keywords: Component-oriented language, modeling, architecture, programming, separation of concerns, decoupling, inheritance, reflection, reification, COMPO

Résumé

L'ingénierie des logiciels a base de composants, produisant du logiciel en assemblant des composants sur « étagere » et « prets-a-l'usage », promet la réduction des couts au cours du développement, la maintenance et l'évolution d'un logiciel. La période récente a vu la production d'un ensemble tres important de nouveaux résultats dans ce domaine. Comme le terme «composant» a un sens assez général, cet ensemble englobe de nombreuses recherches ayant des objectifs différents et offrant divers types d'abstractions et mécanismes. Cependant, une idée générale communément admise consiste a modéliser les logiciels avec des composants organisés en architectures, puis générer du code a partir de ces descriptions abstraites. Ceci est une bonne idée, mais la question qui se pose consiste a savoir quel langage est le meilleur candidat pour le code généré. Dans la pratique actuelle, la phase de conception se déroule dans le monde des composants alors que la phase de programmation se produit dans le monde des objets. Il semble aussi que les langages et technologies utilisées dans le développement a base de composants ne sont que partiellement a base de composants. Notre premiere revendication consiste a dire qu'il est important d'utiliser les langages a composants pour écrire du code exécutable, simplement parce que les artefacts a base de composants d'origine (comme, les besoins ou les architectures) ne disparaissent pas au moment de l'exécution, rendant les programmes plus compréhensibles et réversibles. En faisant cela, il est alors possible d'imaginer que la conception (modélisation) et la programmation peuvent etre effectuées au meme niveau conceptuel et pourquoi pas en utilisant le meme langage. Généralement, les objets sont presque toujours choisis pour implémenter les conceptions a base de composants. Par ailleurs, il est vrai que c'est sans surprise les objets qui sont utilisés pour implémenter des conceptions a base de composants ; un objet étant certainement l'entité exécutable la plus proche d'un composant tel que c'est compris aujourd'hui. Par contre, ils sont proches mais il ne sont pas exactement les memes. Notre deuxieme revendication est qu'il est possible d'atteindre des langages de programmation par composants en apportant des modifications souples aux langages a objets. Suivant ces idées, nous présentons dans cette these un exemple d'un nouveau langage pur de modélisation et de programmation par composants, nommé COMPO intégrant d'une maniere simple et uniforme, les concepts de base pour la description et l'implémentation des composants et des architectures a composants: composants, ports, services et connexions, et les mécanismes nécessaires suivants: l'instanciation, l'invocation de service, la composition et la substitution. Nous soutenons également que la description des composants, leurs architectures (structures) et leurs services (comportement) gagneraient (comme le font les descriptions d'objets) a utiliser des descriptions différentielles qui se basent sur un mécanisme d'héritage. En conséquence, nous proposons une spécification et une implémentation d'un systeme d'héritage en prenant en compte une politique de spécialisation covariante et un mécanisme de sub-

stitution dédié. Nous affirmons enfin que faire un tel langage totalement réflexif ouvrira une nouvelle alternative intéressante (dans le contexte des composants) pour n'importe quel genre de modele ou de programme de vérification ou de transformation d'architecture. Nous revisitons quelques solutions standards pour obtenir une réification a composants originale pour construire un méta-modele exécutable conçu sur l'idée du «tout est un composant». Une implémentation complete du prototype du langage COMPO a été réalisée et est décrite dans cette these.

Mots clés: Langage à composants, modélisation, l'architecture, la programmation, la séparation des préoccupations, découplage, l'héritage, la réflexion, la réification, COMPO

CHAPTER 1

Introduction

To program is to understand.
Kristen NYGAARD.

Preamble

*This chapter introduces the context of our research. The context is software engineering and more specifically Component-Based Software Engineering (CBSE) which study development of reusable components (*development for reuse*) and development by assembling reusable components (*development by reuse*). We explain the problems regarding the existing component-based approaches and the strategies they are using to build component-based software. In this context, we place our approach and the solutions offered. We finish this chapter with presenting characteristics of the contributions and organization of the document.*

1.1 Context: Component-based Software Engineering

EVOLUTION, the ubiquitous process in every science discipline goes hand in hand with complex problems and development of their solutions. It has been proven that a very helpful approach to solve a complex problem is to identify and solve its sub-problems, ergo the *Divide and Conquer* principle. The strategy of constructing a problem's solution from solutions of its sub-problems has been widely adopted by industry in many domains. Hence, for example, we drive cars assembled from reusable and substitutable parts, called **components**, like an engine, wheels, doors, etc.

The process of seeking a viewpoint which simplifies a complex problem is commonly known as *abstraction*. History shows that there are always more viewpoints of a problem and although a found viewpoint, *i.e.* an abstraction of a problem, might be fundamentally wrong, it still can be used to approximately solve the problem. For example, gravity laws as they were described by Isaac Newton are a very good abstraction of weak-field gravity and slow speeds, but they are fundamentally wrong, when used for solving problems in a context of strong-field gravity and high speeds (close to the speed of light), as shown by Einstein. The quality of an abstraction depends on its ability to describe a system that solves a given problem in an accomplishable way.

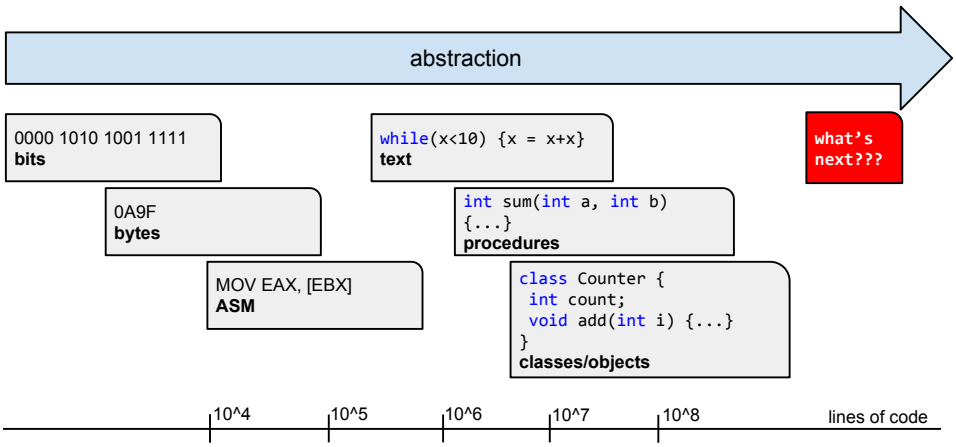


Figure 1.1 : Growing complexity of solutions, here mesured in terms of Lines of Code (LoC), forces the evolution of computer programming languages

Evolution in software development industry, enforced by increasing complexity of developed systems [Kriens, 2012], has driven the creation of modern software in terms of objects and classes and not by assembling individual bits or bytes, like it was usual less than 6 decades ago. Bits or bytes were good abstractions for programming arduous calculations, but due to the amount of information being manipulated and due to the evolution of business processes by new software, it was too expensive to keep using these abstractions for programming present complex systems. Therefore they were replaced by more convenient abstractions being able to describe solutions for modern problems in manageable ways. Figure 1.1 shows how the growing complexity of solutions (measured by terms of lines of code) forced the evolution of computer programming languages. The process can be summarized into the following statement: "*When things got too complex, it is time to increase abstraction level.*"

Year	Operating System	LoC (Million)	Year	Operating System	LoC (Million)	Year	Operating System	LoC (Million)
1994	Windows NT 3.5	7.5	2000	Debian 2.2	57.5	2003	Linux kernel 2.6.0	5.2
1996	Windows NT 4.0	11.5	2002	Debian 3.0	104	2009	Linux kernel 2.6.29	11
2000	Windows 2000	29.2	2005	Debian 3.1	215	2010	Linux kernel 2.6.32	12.6
2001	Windows XP	45	2007	Debian 4.0	283	2010	Linux kernel 2.6.35	13.5
2003	Windows Server 2003	50	2009	Debian 5.0	324	2013	Linux kernel 3.6	15.9

Table 1.1 : Growing complexity of software illustrated in terms of lines of code (LoC) in case of OS

Development of software projects with millions of lines of code (*cf.* Table 1.1) is no longer a matter of a single developer, it is not in ones powers to be knowledgeable enough to confidently change every part of such projects. Thus, the frequent practice is to assign one or more parts of the system to a developer. To understand how a part interacts with the rest of the system becomes crucial information for the developer of the component in order to effectively evolve the component. For example, when modifying the invariants of a data structure, a developer must discover what code within and outside the part relies on those invariants, and make appropriate modifications to that code. Understanding how the parts interact is especially difficult in many modern systems, which communicate indirectly through shared data structures, dynamic dispatch, and events. To evolve these programs effectively, an engineer often needs an abstraction of the possible run-time types and aliases of each element involved in the change. Such abstractions are difficult to gain, and if they are incorrect, engineers are likely to inject defects as they evolve the software system [Aldrich, 2003].

The complexity has been the engine of component-based research with a goal to develop, manipulate and reuse software parts that make up these complex systems. Component-based Software Engineering (CBSE) is an approach that uses Component-oriented Programming (COP)[1] to develop reusable components (*development for reuse*) and to assemble software from these reusable "off-the-shelf" components, connected together into various kinds of architectures (*development by reuse*).

In the sense of seeking for an appropriate abstraction to manage the growing complexity, **CBSE takes the notion of software component and uses it as the abstraction which makes development of complex systems easier and manageable**. The vision of reusable and connectable software pieces made of other reusable and connectable software pieces is consistent with the divide and conquer principle. Moreover, the well-established methodologies and techniques used in other engineering domains, like electric-circuits design or product lines strategies, can be taken and adapted for the purposes of CBSE.

As pointed by Szyperski in [Szyperski, 2002], CBSE opens the possibility to establish a component market, similar to applications markets like AppStore, or Google Play Store, where developers store their components and other developers use the stored components to design new components or to assemble final products. In contrast to applications markets, the consumers of this component market are developers and not final users. Software evolution then would became a matter of updating, because it is easy to substitute a component with a new (more sufficient, effective, etc.) one. Also on demand applications would became real, as it is possible to dynamically assemble an application. For example a demanding customer may require a graph application with 3D rendering component while a graph application with 2D rendering component will be enough for a regular customer. As long as the 3D a 2D rendering components are substitutable, the rest of the application can be reused

[1] COP is a programming technique and paradigm producing reusable components as the output of coding process.

and the two customers can be satisfied easily.

1.2 Limitations of the Existing Approaches

The existing component-based approaches have opened the door for the development of component-based software. They have changed the way distributed applications are developed. In particular, they enable the large-scale deployment of applications in distributed heterogeneous environments, by making transparent the distribution, security and many other non-functional aspects and enabling the developers to focus on application concerns.

The approaches differ in many aspects, for example: in the way they define a component (runtime vs. design-time entity, made from objects or not); whether or not they separate an external contract definition from an internal composition definition; or in the way they define a component's behavior. In this work we divide the approaches into three global categories according to strategy they use to construct a final solution.

The *generative strategy* uses high-level abstraction design models as conceptual tools for managing the complexity of large software systems. These models specified in Architecture Description Languages [Medvidovic et Taylor, 2000] (ADLs) describe the high-level organization of a software system as a collection of components, connections between the components, and constraints on how the components interact. The intent of these models is to communicate to an entire engineering team part of the global knowledge needed to develop and evolve each component of the system. They also aid in the specification and analysis of high-level designs. For example, an architectural model can be analyzed to prove that the design invariants described by architectural constraints are satisfied. Once the architecture design stage of development cycle is finished, the generative strategy takes the formal description of a designed architecture and generates code skeletons using an a generator specific for an implementation language. This implementation-language independence property is another advantage of the approach as the same abstraction layer can generate code for different machines, taking into account the heterogeneity of platforms.

However, this may cause problems in the analysis, implementation, understanding, and evolution of software systems, because consistency between architecture design and final code is not guaranteed. While the architecture design may be analyzed for certain properties, it is difficult to know if the properties hold in the implementation of the design. In addition, even if a system is initially built to conform to its intended architecture design, as the system evolves to address new requirements, its design may become inconsistent with the original architecture design over time. This inconsistency causes problems for engineers working with the system, making it difficult to understand parts of the system in isolation, and causing program errors when engineers rely on their inaccurate architectural models. In summary, inconsistency between architecture and implementation pervades existing systems, causing problems both in human reasoning and automated analysis of programs.

The *framework strategy*, represented by COM+ [Microsoft, 2012], Enterprise JavaBeans [Oracle, 2012], CORBA Components [OMG, 2012], Fractal [Bruneton *et al.*, 2006], Spring [GoPivotal, Inc., 2013], OSGi [OSGi Alliance, 2012], FraSCAti [Seinturier *et al.*, 2012] etc. provides component models and development frameworks for building component-based distributed applications. They all employ a similar global system design providing a separation between the functional aspects of the ap-

plication, which are captured by the components, and the non-functional, technical concerns, which are captured by the containers. Run-time containers provide capabilities to dynamically lookup, load and release components making it possible to develop adaptable distributed component-based systems.

As pointed by [Crnkovic *et al.*, 2011], most component models classified under the framework strategy use mainstream object-oriented programming languages `C++`, `Java` and `C#` for the implementation stage. By following the programming style guidelines of each particular framework a developer is able to design and implement components or create applications by assembling off-the-shelf components. These programming style guidelines often prohibit common programming idioms such as data sharing making the developer's life harder as it tends to be very difficult for them to avoid messing with the system's integrity for their own convenience. Another problem is that the respect of the programming style is not mandatory. The implementation languages do not treat component related concepts, like required interfaces or composition, explicitly separating design development stage from implementation stage, causing that the original component based design may vanish during implementation. For example, in most cases, the connection between objects in object-oriented programming languages are implicit in the implementation code, making it hard to verify that systems have, indeed, the intended architecture with explicit connectors.

While the previous strategies use Domain Specific Languages (DSL) and code generators or programming style guidelines and run-time support systems to produce a software system, they fail in verifying full conformance between a rich architectural specification and an implementation in a general-purpose programming language. The lack of automated conformance checking seriously compromises the benefits of architecture during implementation, testing, and software evolution.

The last approach, the **component-oriented language (COL) strategy**, is an evolution compatible with both previous approaches. It operates in languages design domain and states that the more natural way to develop component-based software systems is to use a single programming language that allows doing so in the first place. Such programming languages should have a primitive support for both component definition, and composition (building components by assembling smaller components).

> "*[...] we feel that there is need for pure component languages. These languages are needed to provide a component developer with a clean and concise vocabulary and semantics for building and composing components.*" [Wuyts et Ducasse, 2001].

One of the main illities CBSE brought to software is modularity. In practice, Object-oriented programming (OOP) also brought some modularity in code, but has some limitations to fully address this quality attribute. Although it is possible to capture a component-based architecture in an object-oriented language, the problem is that the capture is implicit, making it hard to reveal original design intentions later. For example, the following Java code snippet models a very simple text-editor component.

```
public class TextEditor {
  private ISpellChecker sc;
  public TextEditor() {  }
  public void setSpellChecker(ISpellChecker sc) {...}
  public ISpellChecker getSpellChecker() {...}
  ...
}
```

The global semantics of the `sc` attribute with the getter and setter operations is: "*a text-editor requires a spell-checker*". Unless users of such an editor read the editor's documentation or its code, they are not aware of the fact that the editor requires a spell-checker. The information is not explicit. The example illustrates the need for a pure component-based language which treats these concepts explicitly.

Component-oriented Programming Languages (COPLs) like ACOEL [Sreedhar, 2002], ArchJava [Aldrich *et al.*, 2002], ComponentJ [Seco *et al.*, 2008] or SCL [Fabresse *et al.*, 2008] give solutions for the previous problem by allowing developers to express full description of executable components.

Moreover, using different concepts for design and implementation stages of development opens a gap which makes architectural reasoning (*i.e.* the action of thinking about architectures in a logical, sensible way) complicated. For example, architecture should show all of the components that could possibly communicate with a given component. An engineer who is enhancing that component can effectively use this knowledge to make sure that the enhanced component interacts properly with all the existing components in the system. In order to enable architectural reasoning about an implementation, the implementation must conform to its architecture. A system conforms to its architecture if the architecture is a correct abstraction of the run-time behavior of the system. However, an engineer who cannot trust the architecture to be complete must fall back on more labor-intensive techniques for finding the other interacting components, or else risk introducing defects into the code.

Recent experience [Fabresse *et al.*, 2012 ; Aldrich *et al.*, 2002] with COLs showed that it is possible to bridge the gap between implementation and design and that in the same time we obtain better code-level decoupling in component-based programming than in object-oriented one. Architecture implementation conformance is automatically guaranteed, because there is no separation between the design and implementation stage.

The above mentioned approaches provide means to design component-based software, but they do not address very well maintenance and evolution of such software. For example, the generative approach is forced to re-generate implementation of an architecture design every-time the design changes. While smaller changes are manageable, larger ones require wider refactoring leading to possible architecture-implementation inconsistencies. Managing software evolution and increasing productivity are the main objectives of Model-driven engineering (MDE) approach.

Transformations are one of the essential principles of MDE needed for enhancing primary models into final software products [Carrière *et al.*, 1999]. Because MDE support is not present in the above discussed approaches, transformations have to be described in third-party languages [Sánchez Cuadrado, 2012] making it difficult to apply a transformation on a model in a

straightforward way. We believe that the problem lays in fact that the solutions do not pay enough attention to reflection which is a basic need for doing transformations.

Recently the Models@runtime [Blair *et al.*, 2009] stream addresses this by proposing the reflection layer and by considering it as a real model that can be uncoupled from the running architecture (e.g. for reasoning, validation, and simulation purposes) and later automatically re-synchronized with its running instance. For example Meta-ORB [Costa *et al.*, 2006][Provensi *et al.*, 2010] proposes the design time use of models to generate middleware configurations, and, at run-time, the use of these same models as the causally connected self-representation of the middleware components that is maintained by the reflective meta-objects for the purposes of dynamic adaptation. However, the approach still suffers from the disadvantage of separating a model from implementation, making it difficult to ensure that the operational semantics defined in an implementation conform to its model architecture.

The problem of not paying enough attention to reflection rises up again when talking about architectural constraints which represent the formal technique for architecture decision documentation [Allen, 1997 ; Monroe, 2001 ; Tibermacine *et al.*, 2010b]. Constraint are used in MDE to specify and verify non-functional quality attributes of models. Examples of constraints include the choice of a particular architectural style or pattern, like the layered style.

When defining component-based software architecture descriptions, architecture constraints are generally intended for the validation of some specific architectural elements (components, in most cases). This limits their potential reuse with architectural elements of other architecture descriptions. In addition, this kind of architecture decision documentation often includes some parts which can be used individually for documenting parts of design decisions [Tibermacine *et al.*, 2010a]. Unfortunately, there is no means to extract these parts, to make them parametrized entities that can be factorized and used in different reuse contexts.

1.3 SCL, the predecessor of our work

In this work we pick up the threads of the research made during the development of SCL (*Simple Component Language* [Fabresse *et al.*, 2008]) which tried to fill the lack of semantically founded and really usable component-oriented languages (COL) by addressing the following problematics: What is a COL? What are the advantages of those languages? How to achieve a COL?

SCL has been built to be: (i) **minimal** because all its abstractions and mechanisms answer to an identified need; (ii) **simple** because these abstractions and mechanisms are of a high-level; (iii) **detailed** because it targets a lot of crucial points usually forgotten by other propositions such as self-references, arguments passing based on connections or considering base components (collections, integers, etc) in a unified world; (iv) **dedicated** to CBSE because it integrates the two key points that were identified: decoupling and unanticipation. The core of SCL is built upon the following concepts: component, port, service, connector, glue code, and the following mechanisms: port binding and service invocation. All of this is mixed into a language, in which an expert programmer can develop independent components, *design for reuse*, and a non expert programmer can develop applications by connecting previously developed components, *design by reuse*.

SCL applies the class/instance approach and it clearly distinguishes these two concepts. A com-

ponent is a run-time entity and it is an instance of a component *descriptor*. Component descriptors are written by the component programmer in order to create off-the-shelf reusable pieces of software while the software architect creates an application by choosing and instantiating some component descriptors and then connecting components (*i.e.* instances).

The unique communication protocol is built upon unidirectional named ports which allow the programmer to group some services in a set and require or provide this set via required resp. provided ports. Components communicate by *service invocation* through their ports. Ports help the programmer to group related services and then defines view points or security policies. Required ports define view points for the component on its environment while provided ports define view points on the component for its environment. A port also defines a security policy because a component that communicates with another component through one of its ports can only access the services accessible through this port. The programmer defines *services* in the component implementation and chooses to provide some of them through the provided ports of the component.

Areas where SCL might be improved are: explicit architecture description and reflection. SCL is hard to use as an architecture description language, because it focuses more on the functional aspect of components then on the specification aspect. This limits the modeling power of the language Missing reflection level makes it impossible to reason about an architecture design described by a SCL program or to apply automatic transformations needed in MDE development for enhancing an initial design to the final product. In the following we describe a reflective COL based on the outcomes of the SCL research which aspires to be even more detailed and dedicated to CBSE by taking into account explicit architectures and MDE, while, in the same time, keeping the language minimal and simple as much it is possible.

1.4 The problematic of the thesis

This dissertation presents the design and implementation of a reflective component-oriented language named COMPO. This section describes the problems we aim to handle in the work. The global objective of COMPO's design is to try to preserve and improve the qualities of SCL (the predecessor). In this sense, COMPO strives to go further in support for the modeling aspects of CBSE, by making it easier to use techniques and methods designed for ADLs and to perform transformations and verifications typical for MDE.

The study of existing CBSE approaches, presented in the following chapter (*cf.* Chapter 2), convinced us that component-based software should be developed by use of component-oriented programming languages which provide a development continuum by supporting both the design and implementation stages of the development process. In the same time, we would like to have the advantage of explicit architecture description and the advantage of run-time adaptability that we mention in Section 1.2.

The following list presents the problems we plan to handle and answers the question: "How do we want to do that?"

To program and design components using the same language We want COMPO to be a pure component-oriented language (COL) in which it should be possible to design and implement

reusable components. The purity means that we try to not to build just an extension for an existing programming language, which will support component-oriented programming, because it could force developers to choose between components and objects to implement given element of the business domain of their application. Such a COL bridges the gap between design and implementation by eliminating inconsistencies that may occur when two conceptually different languages are used for design and implementation [Fabresse *et al.*, 2012 ; Aldrich *et al.*, 2002].

To handle this: we try to identify the core component concepts and restrict COMPO's design to strictly use those concepts only, making it possible to design and seamlessly implement components in one language.

To model component-based architectures We would like that COMPO users be able to model various component-based architectures in the same way ADLs do so. ADLs explicitly describe architectures of component-based software. With such an explicit approach, the interaction is with the model architecture and not with an intricate sequence of design features. That makes initial understanding on the software easier. But it also means designers working with an explicit architecture of a software system can easily pick up a design where others left off. Much like anyone can open up and immediately continue working. Thus explicit modeling appeals to a variety of audiences: companies with flexible staff; infrequent users; and anyone who is concurrently involved in a large number of design projects. To understand and demonstrate the advantage of being explicit consider the following example:

> "*Let's say we want a person data structure. We can accomplish this by having specific fields, as Listing 1.4 shows. Of course, to make this work, we must define the variables in the person class. Many modern languages provide a dictionary data structure (also knows as a map, associative array, or hash table), so, we could use it to define the person class, using the approach in Listing 1.4. (This is slower, but let's assume this section of code is not performance critical.)*
>
> *Using a dictionary is appealing because it lets you change what you store in the person without changing the person class. If you want to add a telephone number, you can do it without altering the original code.*
>
> *Despite this, the dictionary does not make it easier to modify the code. If I'm trying to use the person structure, I cannot tell what is in it. To learn that someone's storing the number of dependents, I must review the entire system. If the number of dependents is declared in the class, then I only have to look in the person class to see what it supports.*
>
> *The key principle is that explicit code is easier to understand which makes the code easier to modify. As Kent Beck puts, the explicit code is intention revealing. This dictionary example is small in scale, but the principle holds at almost every scale of software design.*" [Fowler, 2001].

To handle this: we design COMPO in a way, it will be possible to model and explicitly describe component-based architectures, *i.e.* it is possible to use COMPO as a regular ADL.

```
class Person {
   public String lastName;
   public String firstName;
   public int numberOfDependents;
}
```

Figure 1.2 : To be Explicit: fields example

```
class Person {
   public Dictionary data;
   /* data["firstName"]; data["lastName"]; ... */
}
```

Figure 1.3 : To be Explicit: dictionary example

To reuse component designs We want to be able to design a new component on the basis of an existing component, i.e. to reuse a component design including both the structure description and the behavior description. There are many ways how to reuse software, like aspects [Kiczales *et al.*, 2001 ; Seinturier *et al.*, 2006], traits [Curry *et al.*, 1982], mixins [Bracha et Cook, 1990] or composition which is already a mechanism in the world of components. The successful reuse mechanism is inheritance, which has proved to be one major cornerstone of software reuse in the OO world, first for the ability it gives developers to organize their ideas on the base of concept classification (a list is a kind of collection, a given architecture is a kind of visitor, ...) which is itself one key of human abstraction power and second for the calculus model that makes it possible to not only reuse but adapt software, by executing an inherited code in a new context (the receiver environment). Despite the success of inheritance, the question of the interest of inheritance-based reuse in the CBSE context has not yet been explicitly nor fully addressed.

To handle this: we design and integrate a component-oriented inheritance mechanism for COMPO.

To support development processes We would like to be able to reason about, transform and verify component-based architectures (possible at run-time), because we believe that such illities are fundamental for MDE. Also, we would like to make it possible for users to customize communication between components or to constraint it. Our previous work [Tibermacine *et al.*, 2011] shows that architectural constraints can be successfully realized as components. The idea is attractive when it comes to verification of architectures qualities, especially, after a transformation was applied. Therefore, we would like to be able to design constraint components and connect them to standard business components to check their qualities, all at both the static-time and run-time and in the context of one language.

To handle this: we shall try to design COMPO as a reflective language by making some aspect of the internal representation explicit and hence accessible from the program. A reflective language or system provides a principled (as opposed to ad hoc) means of achieving open engineering [Blair *et al.*, 1998]. Reflection enables language users to reason about architectures, to

perform model transformations, to examine and modify the structure and behavior of entities (in or case of components) at run-time.

To summarize our dream

It would be nice to have a language in which one can describe a component-based architecture and continuously implement it, then verify and execute or transform the result.

1.5 Characteristics of the contribution

The contributions of this dissertation are as follows:

Uniformity. We propose and describe a component-based meta-model and a reflective description in COMPO of its main component descriptors made executable via a concrete implementation. We present concrete, adapted (first-class descriptors) or new (first-class ports), meta-level solutions for a component-based reification of concepts leading to a "*everything is a component*" operational development paradigm. We tried to design the component-based model compliant to its meta-model and the component-based meta-model compliant to itself. The system is self-described by the explicit definition of the root of the instantiation tree (Descriptor) and the root of the inheritance tree (Component). One consequence of component-oriented reification is that there is now only one kind of entity, component: a descriptor is a component and a meta-descriptor is a true descriptor whose instances are descriptors. This allows a simplification and economy of concepts, which are thus more powerful.

Architecture within Implementation. COMPO tries to smoothly integrate a rich architectural description with a programming language to enforce full structural conformance between design and implementation. COMPO provides architecture description constructs, so that developers can specify an architecture during design and then fill in the architecture with COMPO implementation code. It includes all core concepts of CBSE, *i.e.* components, ports, explicit connections and services. Components are instances, supporting instance-based architectural reasoning, and ownership declarations are used to specify hierarchical relationships between components.

Unique communication protocol. In SCL, sending a service invocation through a port is the only possible way for two components to interact. Nevertheless, we make effort to integrate ownership relation to achieve explicit hierarchical design, when doing so, we proposed a solution based on internal required ports and thus keep the unique communication protocol. Such a protocol is another step towards full communication integrity of COMPO applications.

Openness and extensibility. Reflection provides the necessary levels of openness making the language uniformly accessible by the user. It opens the essential possibility that architectures, implementations and transformations can all be written at the component level and using a unique language. It encourages introspection and indeed adaptation of the underlying structure and behavior of the platform. Reflection ease introduction, possibly dynamic, of different control facilities for components such as non-functional aspects; it allows application designers and programmers to define

important trade-offs such as performance vs. safety; and it makes easier the use of these frameworks and languages in different environments.

Modeling friendly inheritance system. Reuse scheme designed for COMPO is quite innovative in the context of CBSE, because it promotes modeling power with covariant specializations. Using `extends` statement, a new descriptor can be defined on the base of an existing descriptor, such a descriptor is then called a *sub-descriptor*. Sub-descriptors may introduce new ports or extend interfaces of inherited ports; new services and override inherited services and finally a sub-descriptor may extend and specialize the internal architecture it inherits from its parent, ergo, COMPO's inheritance system offers means for both structural and behavioral reuse.

1.6 Structure of the thesis

The remainder of this dissertation is structured to gradually present precise definitions for the concepts informally addressed in this introductory chapter. The contents of the remaining chapters are as follows:

- Chapter 2 presents the state of the art of component-based approach. The general motivation of this approach is presented through three problematic areas of software engineering, namely reuse, distribution and explicitness problems for which the component-based development seems better suited than the object-oriented approach. After presenting the main families of component-based approaches, some approaches are described in detail. Finally, in concluding this chapter, we point on weaknesses of these approaches in the context of component-oriented programming.
- Chapter 3 presents our basic component-based language. Based on the outcomes of Chapter 3 try to identify the main concepts and mechanisms of the component-based approach and then in each section we discuss how we believe COMPO meets these principals.
- Chapter 4 extends the component language presented in Chapter 3 with the component-based inheritance system for structural and behavioral reuse. We motivate the need for an inheritance system by showing cases where an inheritance mechanism is inevitable for reusing the structural definition of component descriptions.
- Chapter 5 describes the self-described component-based meta-model with makes it possible to integrate reflection into the language. We describe how the integrated reflection allow for standard application development, and for static or run-time model and program transformations, all within the context of COMPO language.
- Chapter 6 illustrates the features of our model by means of medium size examples like a HTTP server design, architecture constraints modeling and verification or architecture transformation.
- Chapter 7 presents the prototype implementation in Pharo SMALLTALK starting with the technology choices and meta-model core architecture.

- Chapter 8 draws some conclusions about the proposed component-oriented programming and modeling language and describe some future directions.

CHAPTER 2

Component Models and their Implementations

If a cluttered desk is a sign of a cluttered mind, of what, then,
is an empty desk a sign?
Albert EINSTEIN.

Preamble

In this chapter, we present a state of the art of the Component-Based Software Engineering approach. Section 2.1 recalls and discusses the main advantages and promises of the component-based approach. We detail two principles: reuse and explicitness. Given the multitude of component-based approaches, we made a selection of those we consider pivotal and we detail each selected approach in 2.2. Section 2.3 presents a comparison of the selected approaches, following a set of criteria that we consider relevant to our contribution. Section 2.4 concludes this chapter by providing a concise overview of the different approaches and shows the need to propose a new component-oriented programming and modeling language.

2.1 Advantages and promises of the component-based approach

RESEARCH works on Component-Based Software Engineering (CBSE) have brought many advances on how to achieve complex software development by reusing and assembling components. CBSE studies the ways reusable pieces of code can and should be described (for example by giving an explicit high or low level description of what they require to achieve what they provide) and in the ways software architectures are thought, specified, described or implemented. The current trend is to explicitly express architectures of software solutions, to reason about them, to verify them and to transform them. CBSE also studies the development of distributed applications which requires an interaction between components deployed on different hosts.

This section presents the three main aspects in which component-based development promises to be better suited than traditional development paradigms such as object-oriented development. These three areas are: (*i*) reuse which directly reduces the cost and time during the phases of development and testing [Szyperski, 2002], (*ii*) distribution which complicates the application code by requiring the integration of specific code for communications and (*iii*) explicitness which eases maintenance of software by making code structure self-explanatory, hence understandable. Each of these three areas is developed in a sub-section with the aim to show the contributions or the promise of component-based approach along this axis.

2.1.1 Reuse

Reuse is one of the major goals of software engineering for its potential to reduce cost and time of software development. With increasing count of lines of code (LoC) the need for reuse grows.

The Don't Repeat Yourself (DRY) principle states that:

> "*Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.*" [Hunt et Thomas, 1999].

When the DRY principle is applied successfully, a modification of any single element of a system does not require a change in other logically unrelated elements. Additionally, elements that are logically related all change predictably and uniformly, and are thus kept in sync.

Many concepts and associated mechanisms were invented to reuse code, such as:

- function and function call
- module and module import
- class and inheritance
- *framework* and *framework* setting
- component and assembly of components.

In all cases, reuse is performed in two steps: the identification and declaration of a bounded parameterizable part, a *box*, and a definition of how to *configure* and *use* the box *later*. For example,

the definition of a function is done with formal parameters (a box) which are substituted by actual parameters in a function call (a later configuration and usage). There are generally three types of reuse: *black box*, *white box* and *gray box*.

In black-box reuse, the internal of boxes are hidden. The only possible way a user can configure a box is through a pre-defined external interface. As long as the external configuration interface of a box has not been changed, the box can be easily changed by another with the same or compatible interface. For example, without changing its external specification (signature and semantics), the code of a function can be altered, without the need for changing all programs using the function. Thus, in this setup, functions are black-boxes, because theirs internal implementation is not accessible in the use-time.

At the opposite, the white-box reuse type reveals internals of boxes. In the use-time, a box can be configured either through external interfaces or directly through its definition (implementation.) On the one hand, revealing internals has the advantage of making easier, because all information about a box is accessible, it also provides better configuration opportunities. On the other hand, this internal revealing may cause problems. Consider the following example: Zend Framework[1] is an object-oriented web application framework. Its configuration is made explicit through a set of abstract classes. A user of the framework can configure the framework by implementing specialization classes that define abstract methods of the abstract classes. The definition of the abstract methods is a white-box reuse. Indeed, as the framework evolves, new abstract methods are introduced and existing may be removed or altered, i.e. interfaces of the abstract classes change. Client programs that implement specialization classes for the abstract classes directly by definition of abstract methods may not work when a new version of the framework is about to be used.

Reuse is facing a paradox. To increase the potential for reuse, the black-box approach provides better solutions, because its boxes are more independent from a usage context. However, for use in a given context, mechanisms for setting and adjusting boxes may be necessary, therefore it seems that the white-box approach should be favored. The gray-box reuse type is an intermediate level between the previous two forms. The implementation details of a grey-box can be known or disclosed to understand its realization but cannot be modified by its users and cannot be configured through its interfaces. For example, by specifying the places where a component may be varied (*e.g.*, extension or adaptation points), it is possible to avoid unstable implementation dependencies. That is, a supplier of a component should guarantee that variation points remain invariant in subsequent releases. Such a specification can be seen as a *type* definition from which concrete implementations can be derived all conforming to that *type*.

For a long time it was claimed that object-orientation was the solution to reusability of software. Object-orientation indeed enabled the development of reusable class libraries, such as the Standard Template Library [Stepanov et Lee, 1994] or Foundation Classes for Java or C++. These class libraries provide particular type constructors and API, such as sets, lists, hash tables and so on; however, software reuse in the large [Emmerich, 2002] has never been achieved by object-oriented development.

Reuse of objects is hampered by the large number of fine-grained classes generated during object-oriented modeling that are entangled in a system of association, aggregation and generalization re-

[1] http://framework.zend.com

lationships. The large number of dependencies makes it difficult to take classes out of the context in which they were developed and reuse them elsewhere.

> " *[...] class-based languages encourage the reuse of class definitions through extension, but they do not permit the reuse of a class extension in disjoint parts of a class hierarchy.* " [Flatt, 2000]

In fact, a class is not independent of context as it is coupled to the hierarchy in which it was defined. The problem of *implicit* coupling [Briand *et al.*, 1999 ; Peschanski *et al.*, 2000] results in the fact that in the code of the methods, it is possible to instantiate or use external elements without making these links explicitly through interfaces, *i.e.* the black-box reuse. Figure 2.1 shows an example of implicit coupling between two objects. Each instance of the class `A` will use its own instance of the class `B` to fill functionality `bar`. The problem is the `b=new B()` statement which is inside the class `A` while it should be decided outside. This coupling between objects is implicit because it is embedded in the source code which is not always visible, and it is as strong as non-editable.

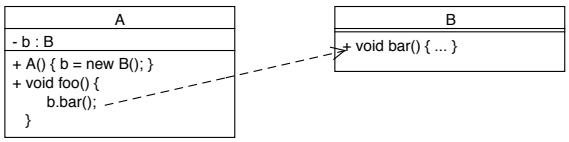


Figure 2.1 : Strong and implicit coupling between two classes

A step towards further decoupling is that the constructor of the class `A` possesses a parameter of type `B` to initialize the value of the `b` private attribute. When instantiating the class `A`, then it would be possible to pass back an instance of the class `B`. The coupling would be made when instantiating the class `A` and not statically in the code. This decoupling is still not satisfactory because it does not allow the setting after `A`'s instances were instantiated. This could be solved with accessors (`get` and `set`), an explicit interface and sub-typing, *see* Fgure 2.2. The class `A` in this example uses the interface `IB` to type the attribute and set accessors. The interface `IB` explicitly declares what features are required and therefore reduce coupling between the class `A` and a particular class implementing the interface, such as `Bimpl`.

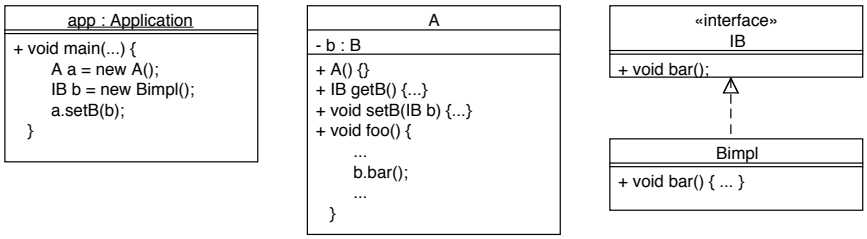


Figure 2.2 : Low and explicit coupling between two classes

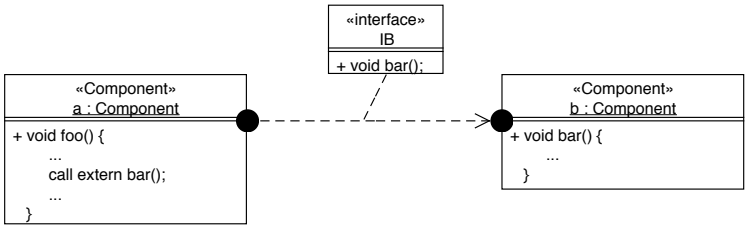


Figure 2.3 : Explicit and low coupling between components

Although it is possible to decouple the objects with an object-oriented language, it requires the use of programming conventions that are not always respected by the programmers. Figure 2.3 shows a schematic example of the component approach which enforces components decoupling and specification of explicit communication interfaces. In this example, the component `a` calls (`call`) a `bar` service it does not define (`extern`). This call will be handled by another component, here `b`, connected to the component `a` later.

Components overcome this problem and provide more easily reusable and more coarse-grained units of code that provide one or more well-defined interfaces. More importantly they provide mechanisms to assemble and configure systems without requiring hard-core programming skills. Thus, in CBSE, we hope to achieve a massive code reuse and even markets of components [Szyperski, 2002] like `componentsource.com`. However further work is needed for simplifying reuse of components.

2.1.2 Distribution

In modern software it is no longer true that an application runs entirely in one address space. Software tend to be distributed. A distributed system is a mechanism in software for normalizing the method-call semantics between application entities residing either in the same address space (application) or remote address space (same host, or remote host on a network.) The need to decouple the business part and the technical part of an application (such as remote communications, security, etc..) has led to the development of *middleware*.

Middleware (*cf.* Figure 2.4) is an intermediate layer between the operating system and the application layer that aims to provide a uniform and transparent view to communicating applications by hiding distribution, heterogeneity of systems, hardware and communication protocols.

Using a middleware ensures independence between the business application code and the technical code that is facing many problems such as low-level concurrency; marshalling (serializing and deserializing the arguments and return values of method calls) or distributed garbage collection. Middleware usually provides a common set of high-level services (also called *not functional* services) as concurrency control, transactions and security. The middleware objects or ORB (*Object Request Broker*) allow a remote object method invocation. Several middleware exist, certainly, the well known is the CORBA standard (*Common Object Request Broker Architecture*) of the OMG (*Object Management Group*).

In the past decade, architectures of distributed objects have evolved into distributed component

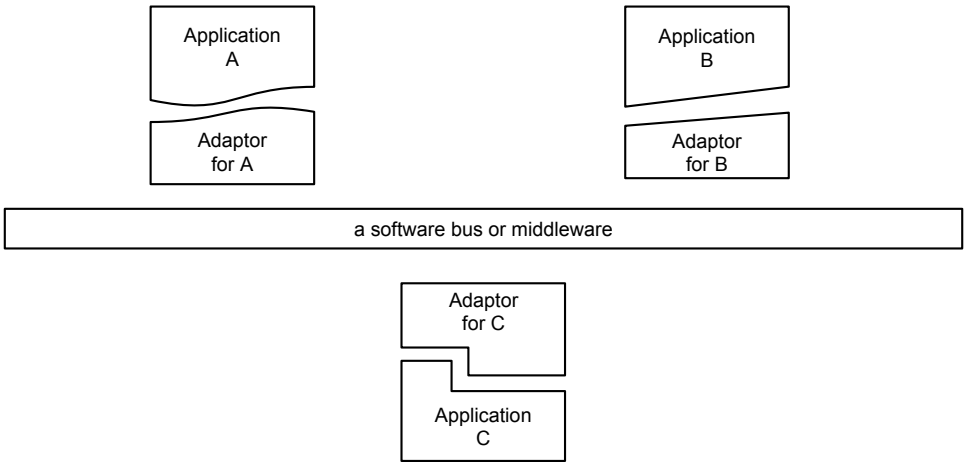


Figure 2.4 : Principle of middleware

architectures. For example, the OMG has developed *CORBA Component Model* [OMG, 2012] (CCM) which builds a component model on top of CORBA objects. This movement towards components was particularly motivated by the problems of coupling discussed in Section 2.1.1 (reuse), but also by the difficulty to separate the business and technical or non-functional concerns using an object-oriented approach. With components, the code for using non-functional services is not mixed with the business logic code.

The communication between the business layer and technical layer is controlled by a third entity in the middleware platform. This third entity calls the business code when necessary. This principle of outsourcing, sometimes called *inversion of control*, is similar to the operation of *frameworks* where the code written by a programmer is called by *framework* code . An example of such a third party entity is an EJB *container* which supports the execution of an EJB component, as detailed in Section 2.2.2.

Although the ideas of CBSE have already taken shape in a number of component systems – both industrial and academic. The industrial systems (represented mainly by EJB [Oracle, 2012] and CCM [OMG, 2012]) are oriented on providing a stable and mature run-time, even at the cost of sacrificing the option of building component hierarchies and other advanced features (such as multiple communication styles, behavior description, etc.) [Bures *et al.*, 2006]

Moreover, middleware platforms are built in standard programming object-oriented languages forcing the users of these platforms to develop their components by use of the same languages. Once again, the issues raising when a non component-oriented language is used for the design and implementation of a component-based system (*cf.* 1.2) limit comprehensibility of such software making it harder to maintain and evolve.

2.1.3 Explicitness

Knowledge sharing and understandability are key factors for the development of large-scale software projects. Making thoughts and ideas "visible", *i.e.* explicit, is the way to support these key factors. The value of making things explicit is that one can "see" what others are thinking and what are the original design intentions.

Where design counts is often not in how the software runs but in how easy it is to change. This drive towards changeability is why it is so important for a design to clearly show what the software does and how it does it. After all, it is hard to change something when you cannot see what it does. An interesting corollary of this is that people often use specific designs because they are easy to change, but when they make the program difficult to understand, the effect is the reverse of what was intended.

While the size of applications constantly grows, it is becoming difficult to understand, maintain, and test programs written in standard programming languages. Several code visualization techniques [Langelier *et al.*, 2005 ; Ducasse *et al.*, 2006 ; Wettel et Lanza, 2007] have been proposed to ease understanding and evolution of programs' architectures. It is becoming evident that although it is possible to capture an architecture in an object-oriented language, the problem is that the capture is implicit, making it hard to reveal original design intentions later. In fact, object-oriented languages and the current trend of modeling software architectures using UML is criticized as being inadequate for the development of large-scale applications [Garlan et Kompanek, 2000].

The component-based approach with its ability to express software architectures [Shaw *et al.*, 1995] in terms of interconnected components, seems to offer a vision suitable for large-scale programming. The high-level languages dedicated to the architecture description attempt to explicitly describe the structure and behavior of software are the starting point for understanding software's architecture as a whole and thus offer high-level techniques adapted to large-scale architectures such as the model for evolution problem in software architecture SAEV [Oussalah *et al.*, 2006] or a mechanism for automatic replacement of a component while preserving the qualities of an original architecture [Desnos *et al.*, 2007]. In addition to specifying the structure and topology of the system, the architecture shows the intended correspondence between the system requirements and elements of the constructed system. It can additionally address system-level properties such as capacity, throughput, consistency, and component compatibility. Architectural models clarify structural and semantic differences among components and interactions. Thus components can be re-used in different contexts [Shaw *et al.*, 1995] or be composed to define larger systems.

2.2 Presentation of the main Component-based approaches

In Section 2.2.1 we try to identify and present the main families of approaches and the reasons that drive us when choosing the representatives of each family.

2.2.1 Families of the component-based approaches

There are many ways how to classify component-based approaches, for example [Crnkovic *et al.*, 2011] focus on component-frameworks and [Medvidovic et Taylor, 2000] on ADLs. Sometimes an

Domain	AUTOSTAR	BIP	BlueArX	CCM	COMDES II	CompoNETS	EJB	Fractal	KOALA	KobrA	IEC 61131	IEC 61499	JavaBeans	MS COM	OpenCOM	OSGi	Palladio	PECOS	Pin	ProCom	Robocop	RUBUS	SaveCCM	SOFA 2.0
General purpose				X		X	X	X		X			X	X	X		X		X					X
Specialized	X	X	X		X				X		X	X				X		X		X	X	X	X	

Table 2.1 : General purpose and domain specific component models [Crnkovic *et al.*, 2011]

approach may belong to more than one category. For example, Fractal [Bruneton *et al.*, 2006] component model is made of the framework for component-oriented development and the Fractal ADL which ease use of the framework. In fact, both could be used separately. One can use the framework without the ADL and another one can use the ADL to describe an architecture and then use a model-specific generator for another component-model. Thus, Fractal is a component-framework and, in the same time, an ADL.

In the previous chapter, we have sketched a way how to divide them into three global families according to the strategy they use to construct a final solution:

Frameworks family provides development frameworks for building component-based distributed applications based on mainstream object-oriented programming languages. Functional aspects of the application are captured by components, and non-functional aspects by run-time containers.

Generative family uses high-level abstraction design models as conceptual tools for managing the complexity of large software systems. Once the architecture design stage of development cycle is finished, the generative family takes the description of a designed architecture and generates code skeletons using a generator specific to an implementation language.

Component-oriented languages family operates in languages design level and states that the more natural way to develop component-based software systems is to use programming languages that allow doing so in the first place.

The classification made in [Crnkovic *et al.*, 2011] divide approaches into general purpose and specialized models. Table 2.1 shows that the distribution between general-purpose component models and specialized component models. Following our classification, all the component models belong to the *frameworks family* as they use mainstream object-oriented programming languages `C++`, `Java` and `C#` to build component-based software. One of the goals of this thesis is to provide a general purpose language for developing component-based systems. Thus, we will consider the general purpose models identified in [Crnkovic *et al.*, 2011] as representatives of the *framework strategy* family. In addition, we chose to study: OpenCORBA for its reflection maturity; Kevoree [Daubert *et al.*, 2012], DynamicTAO [Kon *et al.*, 2000], MetaORB [Costa *et al.*, 2006] as representatives of the Models@runtime [Blair *et al.*, 2009] stream and FraSCAti as an implementation of the SCA [OASIS, 2013] approach. Finally, we also study OSGi [OSGi Alliance, 2012].

The classification of Architecture Description Languages (ADLs) made in [Medvidovic et Taylor, 2000] says that these languages describe the high-level organization of a software system as a collection of components, connections between the components, and constraints on how the components

interact. ADLs belong to the *generative family* because it takes the formal description of the designed architectures and generate code skeletons using a generator specific to an implementation language. This implementation-language independence property is an advantage of the approach as the same abstraction layer can generate code for different machines, taking into account the heterogeneity of platforms. We take the ALDs described in [Medvidovic et Taylor, 2000] as representatives of the *generative strategy* family. In addition, we study UML components package [Cheesman et Daniels, 2000] which provides means for describing component-based architectures and can be used as a model for generating code.

As representatives of the *component-oriented language family* we choose ACOEL [Sreedhar, 2002], ArchJava [Aldrich *et al.*, 2002], ComponentJ [Seco *et al.*, 2008], CLIC [Bouraqadi et Fabresse, 2009] and Bichon [Xu et Ren, 2010] because they all address the implementation stage of a component-based development in a language-level by allowing developers to express full descriptions of executable components within a programming language.

Each following section is devoted to the study of the previous approaches driven according to the following plan:

1. Study plan
   a) Basic Overview
   b) External contract description & Architecture design description
   c) Inheritance
      i. Structural inheritance - the ability to reuse the structure definition, *i.e.* external & internal contracts and architectures
      ii. Behavioral inheritance - the ability to reuse the behavior definition of components
   d) Reflection
      i. Introspection - the ability of a system to observe, and thus reason, about itself comprising the operations defined at the meta-level which examines the data structures of the model
      ii. Intercession - the ability of a model to modify its execution state comprising the operations of a meta-level which change the data structures of the model
      iii. Reification - the method used to expose the internal representation of a system in terms of entities that can be manipulated at run-time

2.2.2 Frameworks family

The frameworks family uses mainstream object-oriented programming languages, such as `C++`, `Java` or `C#`, to build component-based software. By following the programming style guidelines of each particular framework a developer is able to design and implement components or create applications by assembling off-the-shelf components. To ease the application of the programming style guidelines, the models are sometimes accompanied by a model-specific ADL used for generating prearranged source code. The members of this family employ a similar global system design providing a separation between the functional aspects of the application, which are captured by the components,

and the non-functional, technical concerns, which are captured by the container. Run-time containers provide capabilities to dynamically lookup, load and release components making it possible to develop adaptable distributed component-based systems.

SOFA 2

SOFA 2 [Hnětynka et Plášil, 2006 ; Bures *et al.*, 2006] is a component system employing hierarchically composed components. It is a direct successor of the SOFA component model [Plásil *et al.*, 1998], which provides the following features: ADL-based design, behavior specification using behavior protocols, automatically generated connectors supporting seamless and transparent distribution of applications, and distributed run-time environment with dynamic update of components.

From its predecessor, SOFA 2 has inherited the core component model, which is however improved and enhanced in the following way: (1) the component model is defined by means of its meta-model; (2) it allows the dynamic reconfiguration of component architecture and accessing components under the SOA concepts; (3) via connectors, it supports not only plain method invocation, but in fact any communication style; (4) it introduces aspects to components and uses them to clearly separate the control (non-functional) part of components and to make it extensible.

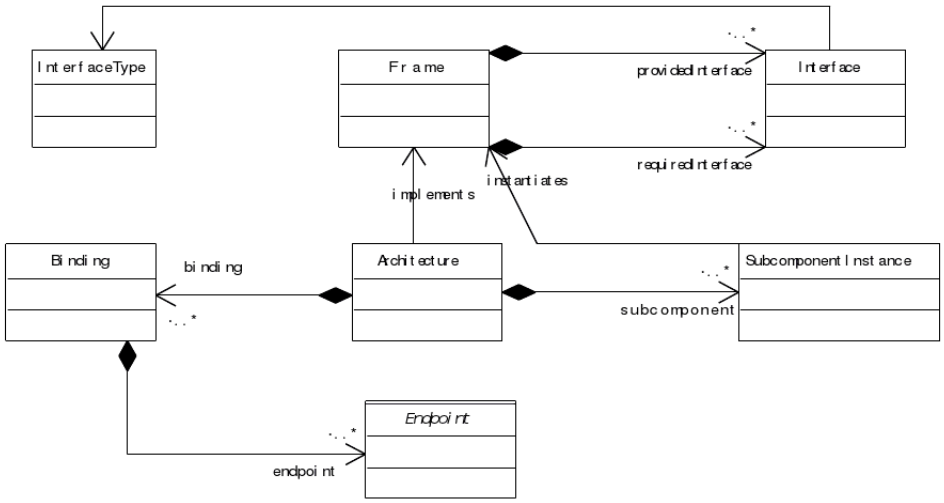


Figure 2.5 : Meta-model of SOFA2

SOFA 2 and all its features are defined using a meta-model (*see* Figure 2.5). The meta-model serves directly for the component specification, which is stored in the repository [Hnetynka et Pise, 2004] and used throughout the application life-cycle. Being a hierarchical model, it allows components to be hierarchically nested. Components can be either primitive or composite. A composite component is built of other components, while a primitive one contains no sub-components. In SOFA 2, a component is an encapsulated entity interacting with other components only via designated provided and required interfaces. A component can play the role of both a black-box and gray-box entity. The black-box role is represented by a component frame, which specifies the set of interfaces, both provided and required, and determines the component's type. As a gray-box, a component is specified

as an architecture that implements a particular component frame (or a number of frames). An architecture of a composite component specifies the sub-components and the bindings between their interfaces. The bindings are performed using connectors that are dynamically generated at deployment time. An architecture of a primitive component stays empty and directly implements the corresponding frame. SOFA defines the CDL (*Component Definition Language*) language, based on OMG IDL to describe interfaces, frames and architectures of components. The language has the ability to attach behavioral descriptions in the form of *protocols* [Plásil *et al.*, 1999].

SOFA CDL provides a single inheritance mechanism for interface specification reuse based on the sub-typing for interfaces. Due to the dichotomy between frames and architectures, SOFA proposes two multiple inheritance mechanisms for specification reuse, one for frames and one for architectures. SOFA proposes to solve the name collisions occurring during the multiple inheritance using explicit renaming. Frames inheritance mechanism preserves a form of sub-type relation in the inheritance hierarchy. Architectures inheritance mechanism has a non-combinational form. There is not a notion of behavioral inheritance.

SOFA supports introspection features in form of so-called control interfaces (a.k.a. controllers). Controllers are not usually accessed by the application logic, but rather by the run-time environment. The control interfaces together with their implementation form a control part of components in SOFA 2 which is modular and extensible. A notion of intercession features is captured by the DCUP extension of SOFA [Plásil *et al.*, 1998]. It proposes an extension of the SOFA model components to support the update of components in a safe way for their execution. DCPU architecture introduces a new notion where components are split into two parts, permanent and replaceable parts, as well as into a functional and a control part. Updates only update the replaceable part of the component, replacing it with a newer version. The updating process is controlled by a component manager, which exists in the permanent part of the component, thus making the component itself responsible for how the updating process is performed. Although SOFA 2 is defined by the meta-model, the entities of the meta-model are not reified, hence not accessible in run-time. Reflection can only be used if it is provided by a target (object-oriented) implementation language.

Fractal

Fractal component model [Bruneton *et al.*, 2006 ; Bruneton *et al.*, 2004] is designed to be used in variety different software branches, *e.g.* middleware, operating systems, information systems and graphical user interface libraries. The main Fractal's design principles are: composite components, shared components to model resources, introspection capabilities to monitor a running system, configuration and dynamic reconfiguration capabilities. The modular and extensible organization allows the use of Fractal in different situations from highly optimized and hardly configurable to less optimized and heavily configurable and dynamic applications.

Fractal components consist of two parts: a controller and a content. The controller is a set of interfaces designated to control behavior, functional and non-functional aspects of a component like introspection, configuration, security and transactions. Controller interfaces may be internal which are accessible from component's sub-components or external which are accessible from outside of the component. Further, controller interfaces are divided in functional interfaces and control interfaces. The functional interfaces are provided or required interfaces, representing functional aspects of

a component. Functional interfaces are equivalent to *remote* interfaces from EJB. Control interfaces that are equivalent to EJB's *home* interfaces are provided interfaces that correspond to non-functional aspect of components such as introspection or configuration. Fractal defines various predefined controllers: attribute controller, binding controller, content controller and life-cycle controller.

The content represents internals of a component. Fractal defines three kinds of components depending on its internals and exposed control interfaces:

- a composite component is a component that exposes a content controller in order to add or remove its sub-components.
- a primitive component is a component that does not expose its content controller, but has at least one control interface.
- a base component is a component that does not expose any control interface.

Fractal allows a component to be owned by various distinct components. Such components are called shared components. They are usually used to represent shared resources.

Creating software using Fractal takes three steps: first, write the implementation code, second, add code annotation with Fractal meta-information, and last, write the linking code using Fractal Architecture Definition Language (ADL) [Leclercq *et al.*, 2007]. The code annotations provide the information about classes that implement components and fields that represent required services, among others. Fractal ADL was created in order to describe the architecture of Fractal components. It is an open, extensible XML based language. The language hides some implementation details, like implementation of attribute, binding or content controllers. Fractal ADL allow users to describe primitive and composite components. Primitive components specify only provisions and requirements in form of provided and required interfaces and a content which is a name of an implementation class. Composite components contain nested sub-components that may be interconnected with bindings.

A definition of components and also sub-components in Fractal ADL may use multiple inheritance mechanism to achieve structural definition reuse. Inheritance is simply an extension of the mechanism that allows adding and overriding component's interfaces, sub-components, bindings, attributes and implementation class definitions. Conflicts resulting from multiple inheritance are solved by linearizing the inheritance graph. Behavior inheritance is not present at the level of ADL, however it is possible to sub-class and override an implementation class in the level of implementation language.

The Fractal component model consists of a framework for the instantiation of components and a set of specifications that a component should or should not implement depending on what control capabilities a component developer wants to offer to users of the component. The control interfaces are special provided interfaces with predefined names organized in levels of control with gradually increasing reflective and introspection capabilities of components:

- lowest level components have no control capabilities, only their methods may be invoked. These components serve only as a component embedding of existing objects.

- the next level provides introspection capabilities of components through a standard interface. This interface allows a user of a component to discover all external interfaces of the component.
- the last level, also called configuration level provides control interfaces to introspect and modify the content of a component that consists of sub-components interconnected with bindings.

The model also allows to perform dynamic reconfigurations of Fractal structures. This is achieved by means of a series of calls to the framework or by using its underlying scripting language FScript.

There are currently two reference implementations: *Julia* [2] (Java) and *Cecilia* [3] (C/C++) and other experimental implementations as *FractTalk* [4] (Smalltalk) or *FractNet* [5] (.Net).

Kevoree

Kevoree[6] [Daubert *et al.*, 2012] is an open-source dynamic component model, which relies on models at run-time [Blair *et al.*, 2009] to properly support the dynamic adaptation of distributed systems.

Models@runtime basically pushes the idea of reflection one step further by considering the reflection layer as a real model that can be uncoupled from the running architecture (e.g. for reasoning, validation, and simulation purposes) and later automatically resynchronized with its running instance. In particular, Kevoree provides a proper support for distributed models@runtime.

In Kevoree, components encapsulate business functionalities. Each component provides a set of functionalities exposed to others. A component also requires functionalities to achieve their own ones. All these functionalities are identified by a port (required or provided) on the component. Components communicate only through their ports.

Components are described by component types which are realized as Java classes. This make it possible to use the standard Java inheritance to achieve structural and behavioral reuse. This also means that Kevoree components are sophisticated Java objects.

Kevoree introduces the *Node* concept to model the infrastructure topology and the *Group* concept to model semantics of inter node communication during synchronization of the reflection model among nodes. Kevoree includes a *Channel* concept to allow multiple communication semantics between remote components deployed on heterogeneous nodes. All Kevoree concepts (Component, Channel, Node, Group) obey the object type design pattern to separate deployment artifacts from running artifacts.

Kevoree supports multiple kinds of execution node technology (*e.g.* Java, Android, MiniCloud, FreeBSD, Arduino, ...).

Kevoree aims at providing advanced adaptation capabilities to different types of nodes:

Level 1 : Parametric adaptation. Dynamic update of parameter values, *e.g.* change of sampling rate in a component that wraps a physical sensor (adaptation of instance properties).

[2] http://fractal.objectweb.org/julia/index.html
[3] http://fractal.ow2.org/cecilia-site/current/
[4] http://csl.ensm-douai.fr/FracTalk/Smalltalk
[5] http://archive.is/rosfp
[6] http://www.kevoree.org

Level 2 : Architectural adaptation. Dynamic addition or removal of bindings or components, *e.g.* replication of software components and channels on different nodes to perform load balancing (adaptation of instances graph).

Level 3 : Dynamic provisioning of types. Hot deployment of component types that were not foreseen before the initial deployment of the system. This allows for system evolution by enabling parametric and architectural reconfigurations, including management of instances for types that are added and managed dynamically (adaptation of types).

Level 4 : Adaptation for remote management. Nodes supporting level 4 adaptation participate in a remote management layer, which supervises less powerful nodes. This layer monitors remote nodes by requesting their current Kevoree model; the layer triggers dynamic adaptation of nodes by sending precomputed reconfiguration scripts to them. This remote adaptation process supports seamless management of less powerful nodes by a more powerful one, which has enough resources to build and evaluate new and appropriate configurations.

The adaptation engine relies on a model comparison between two Kevoree models to compute a script for a safe system reconfiguration; execution of this script brings the system from its current configuration to the new selected configuration [Morin *et al.*, 2009].

Model comparison yields a delta-model defining changes (using CRUD operations) that should be applied on the source model to obtain the target model. planification algorithms use this delta-model as input in order to defined an efficient schedule of the adaptation steps. The delta-model is finally compiled into a Kevoree script. The Kevoree Script language (KevScript for short) is a core language for describing reconfiguration. KevScript is comparable to FScript for Fractal Component Model. Execution of a KevScript directly adapts a Kevoree system, without the need for a full Kevoree model definition. Such adaptation scripts are written by designers, or they can be generated by automated processes (*e.g.* within a control loop managing the Kevoree system).

COM Component Object Model

COM (*Component Object Model*) [Microsoft, 1995 ; Rogerson, 1997] is a component model developed by Microsoft. COM component model is designated to run components within different processes on the same computer. A distributed version DCOM (*Distributed COM*) extends functionality of COM to run components over a network. The last COM version COM+ extends COM with Microsoft Transaction Server - MTS to use transactions, Message Queue Server - MSMQ for asynchronous invocations and other services for improving performance and security. COM specification is not restricted on any platform, however the mainly supported COM platform is Microsoft Windows.

COM components are not restricted on one programming language, they may be written in any programming language whose compiler is able to compile into a binary with an internal structure including virtual tables and function calling conventions as specified in COM specification.

COM uses *Object Remote Procedure Call* - ORPC which is built on top of DCE/RPC [Group, 1997]. To define components a MIDL (*Microsoft IDL*)[7] may be used. MIDL is not directly used by COM, it is used to pre-generate source code with MIDL compiler.

[7] MIDL is an extension of CORBA IDL

COM component model allows to specify only provisions of components in form of provided interfaces. COM provides capabilities for introspection of provisions of components, but requirements must be obtained by programming from a source code. The interface discovery mechanism is implemented through the notion of a special interface called `IUnknown` that must be implemented by every COM component. The purpose of `IUnknown` is actually twofold: (i) it allows the dynamic querying of a component (`QueryInterface()` operation) to find out if it supports a given interface (in which case, a pointer to that interface is returned), and (ii) it implements reference counting in terms of the number of clients using components' interfaces. Reference counting is used to garbage collect components when they no longer have any clients.

A COM interface specifies a set of method signatures and has the following characteristics:

- it has a unique identifier called IID (*Interface IDentifier*), a 128-bit number generated by a pseudo-random algorithm to avoid conflict;
- it is immutable and any changes such as the addition, modification or removal of a method signature, is impossible. A new functionality should be introduced by adding a new interface instead of modifying existing interfaces; This constraint makes managing different versions of the same interface as they must have different identities;
- it inherits directly or indirectly from the interface `IUnknown`;
- it is described in MIDL (*Microsoft Interface Definition Language*).

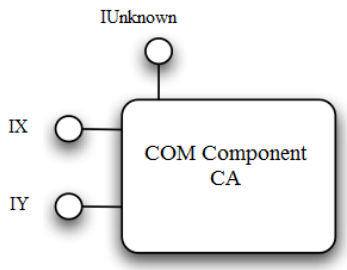


Figure 2.6 : Graphic of a COM object named CA has two interfaces IX and IY representation

COM objects are instances of classes that can be used only through their interfaces. Figure 2.6 shows a graphical representation of a COM object. COM specifies an interface of a COM object must be a pointer (accessible by customers) to an area of the component called node interface.

COM supports two approaches in hierarchical composition:

- *containment*: an owning component reimplements part or all provided interfaces of sub-components. Reimplementation of interfaces may pass calls to a sub-component, whose interfaces are reimplemented.
- *aggregation*: when a client is obtaining an interface from a component in order to invoke component's services a sub-component's interface may be returned and therefore a client works directly with a sub-component.

COM hierarchical composition is not explicitly captured by a kind of ADL, but at source code level. It is actually only a design pattern how to create composed components.

In order to use a COM component, it must be registered in a windows registry database. A GUID (*Globally Unique Identifier*), which is a 128 bit key is used to identify COM components within the registry database. Windows registry database also contains a table to convert a class name to a GUID identifier. COM components are shared within a system therefore any application can use any of registered components. This has a drawback that a replacement of a COM component with a newer version, may cause compatibility problems in other applications that also use the replaced COM component.

COM model does not provide any means for structural or behavioral inheritance in the component level of abstraction. However it is possible to reuse interface definition by the interface inheritance of MIDL.

OpenCOM

OpenCOM [Clarke *et al.*, 2001] is a lightweight and efficient component model based on Microsoft COM model. It includes the binary level interoperability standard, Microsoft's IDL, COMs globally unique identifiers and the `IUnknown` interface. The higher level features of COM such as distribution, persistence, transactions and security are not used. OpenCOM proposes an approach to the design of configurable and open middleware platforms based on the concept of reflection. More specifically, OpenCOM defines a reflective architecture for next generation middleware platforms, supplemented by an open and extensible component framework.

The key concepts of OpenCOM are capsules, components, interfaces, receptacles and connections. Capsules are run-time containers and they host components. Each component implements a set of custom receptacles and interfaces. A receptacle describes a unit of service requirement. An interface expresses a unit of service provision, and a connection is the binding between an interface and a receptacle of the same type.

The general structure of any OpenCOM component is to have a header file for each provided/required interface, and one source file for the implementation of a component.

The *header* file is the place in which a developer defines an interface as a C struct. An "interface" is a struct containing method pointers. Typically developers define receptacles as a receptacle list in a component source file, which is a struct containing a list of pointers to interfaces. A component source file will therefore need to include the header files of any other component interfaces that should be bound to component having receptacles.

The *source* file is the place to define the component's constructor, destructor, and interface method implementations. Inside a component's constructor, we register the interfaces and receptacles of the component with the OpenCOM kernel, and in the process point the interface method pointers at our implementations of those methods.

Each OpenCOM component must inherit the implementation (through *containment* [Rogerson, 1997]) of three standard sub-components (called *MetaInterception*, *MetaArchitecture* and *MetaInterface*). These implement the reflective facilities identified in [Blair *et al.*, 1998] and (respectively) export

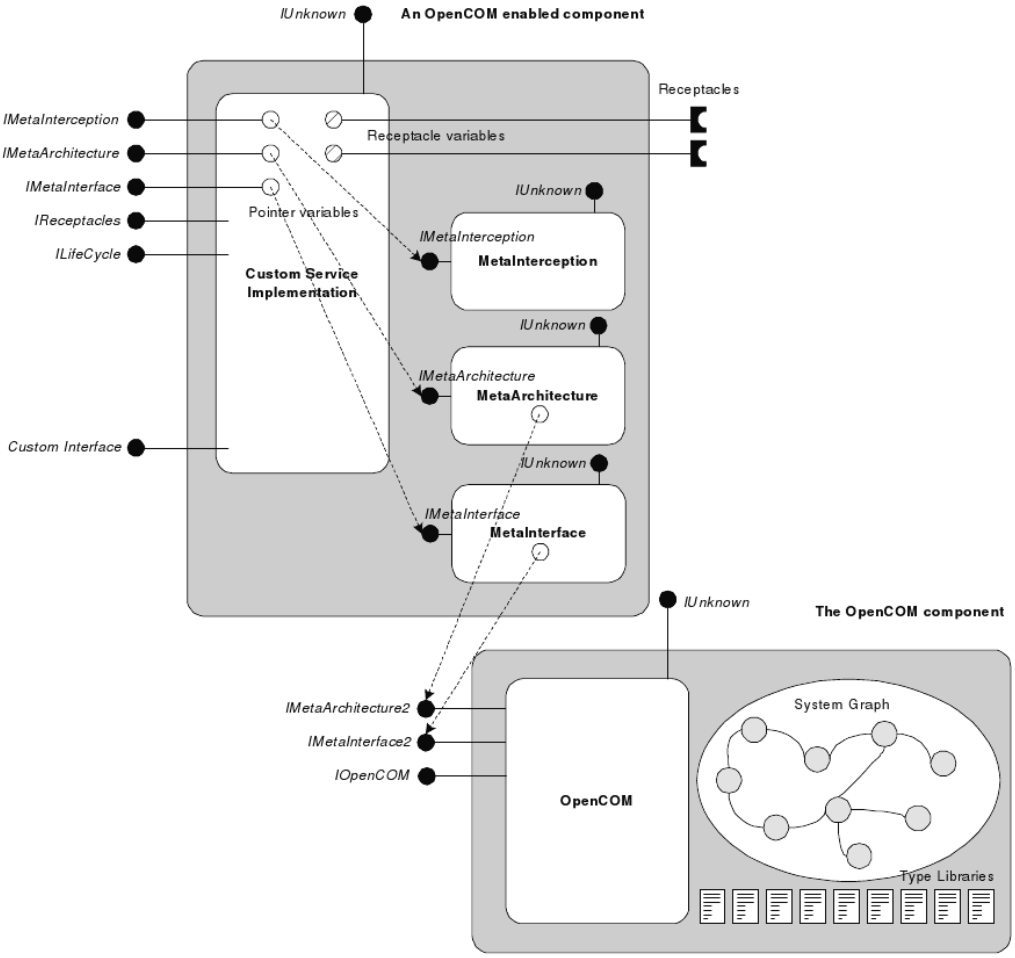


Figure 2.7 : The Architecture of OpenCOM

the following meta-interfaces from the host component:

- *IMetaInterception* enables the programmer to associate (dissociate) interceptor components with (from) some particular interface. Interceptors implement interfaces that contain interceptor methods; these are invoked before or after (or both before and after) every method invocation on the specified interface. Multiple interceptors can be added/ removed at run-time and reordered as desired.
- *IMetaArchitecture* enables the programmer to obtain the identifiers of all current connections between the host components' receptacles and external interfaces. These identifiers can then be used to obtain information about the receptacle/interface/components involved in the connection.
- *IMetaInterface* supports inspection of the types of all interfaces and receptacles declared by the host component.

Figure 2.7 visualizes the component model. It shows an OpenCOM enabled component (top) and the OpenCOM run-time component (bottom). The components' management and meta-interfaces are shown on the left hand side of the OpenCOM enabled component. The three meta-interfaces are linked to the embedded sub-components that implement OpenCOM's reflective capability. Among these interfaces, *MetaArchitecture* and *MetaInterface* are further linked to corresponding private interfaces in the run-time. Also associated with the illustrated component are a component specific interface (labeled "custom interface") and two receptacles. Components can export any number of component specific interfaces and receptacles. The OpenCOM runtime component is shown encapsulating the system graph and type libraries, and exporting the `IOpenCOM` interface.

Similarly to the COM model, OpenCOM does not provide any means for structural or behavioral inheritance in the component level of abstraction.

CORBA Component Model (CCM)

CORBA (*Common Object Request Broker Architecture*) specified by the OMG is a widely used standard for middleware and distributed computing. A part of the CORBA specification called CORBA Object Model (CCM) provides a support for remote procedure calls independently of a communication protocol, programming language, operating system and hardware platform. OMG also provides a set of CORBA Object Services that includes *naming service, trading service, transactions, security, persistence* and others.

Corba uses *Interface Definition Language - IDL* to describe procedures and functions that may be remotely invoked. Mappings of IDL to various programming languages like C++ or Java are defined. This implies possibility to use CORBA with various programming languages.

The version 3.0 of Corba specification introduces CORBA Component Model - CORBA CM, based on CORBA Object Model, which covers a set of software component features. It extends Corba Object Services to provide component specific services for managing, configuring, deploying and interconnecting components.

Component IDL (CIDL), which is an extension of CORBA IDL for defining components. CIDL defines two kinds of components: *basic components* and *extended components*. Basic components serve to simply encapsulate existing CORBA objects within a component. They cannot inherit from other components and cannot specify provisions and requirements. Only attributes are allowed to be specified for component configuration purpose.

Extended components provide a rich set of component functionality. Extended components provide two kinds of provisions and requirements for synchronous and asynchronous invocations. Provisions for synchronous invocations are called *facets* and requirements *receptacles*. Asynchronous provisions and requirements are called *event sources* and *event sinks*. CORBA CM also provides a possibility to define attributes of extended components which are named values, primarily intended for a configuration purpose.

CORBA objects that want to provide direct access to their meta-data have to implement *reflection provider interfaces*. A client can determine whether a given object supports CORBA reflection by attempting to narrow the object's reference to the desired *reflection provider interface*. The *reflection provider interface* supports operations for meta-data retrieval in two formats: XML and an *any* (a

type) containing an instance of an IFR interface description structure.

CCM uses the CORBA object model as its underlying object inter-operability architecture and thus is not bound to a particular programming language. All CCM components support introspection interfaces, which can be used to discover the capabilities of components. The *CCM Navigation interface* (*see* Listing 2.1) defines introspection methods (`get_all_facets()` and `get_named_facets(in NameList names)`) for discovering facets (provided ports). The *CCM Receptacles interface* (*see* Listing 2.2) introspection methods (`get_all_receptacles()` and `get_named_receptacles(in NameList names)`) for discovering receptacles (required ports).

```
interface Navigation {
  Object provide_facet (in FeatureName name) raises (InvalidName);
  FacetDescriptions get_all_facets();
  FacetDescriptions get_named_facets (in NameList names) raises (InvalidName);
  boolean same_component (in Object object_ref);
};
```

LISTING 2.1 : CCM Navigation interface

```
interface Receptacles {
  Cookie connect ( in FeatureName name, in Object connection )
    raises ( InvalidName, InvalidConnection,
             AlreadyConnected, ExceededConnectionLimit);
  void disconnect (in FeatureName name, in Cookie ck)
    raises ( InvalidName, InvalidConnection,
             CookieRequired, NoConnection);
  ConnectionDescriptions get_connections (in FeatureName name)
    raises (InvalidName);
  ReceptacleDescriptions get_all_receptacles ();
  ReceptacleDescriptions get_named_receptacles (in NameList names)
    raises(InvalidName);
};
```

LISTING 2.2 : CCM Receptacles interface

CORBA CM is a flat model though it does not support hierarchical composition of components. Hierarchical composition may be substituted by exposing requirements, that represent sub-component's interfaces, from a component and connecting them to a component that is supposed to be a sub-component.

Structural and behavioral inheritance is not present in the component level of abstraction. In the traditional CORBA object model, interfaces can be extended only via inheritance. To support new interfaces, therefore, application developers must: (1) use CORBA's Interface Definition Language (IDL) to define a new interface that inherits from all the required interfaces; (2) implement the new interface; and (3) deploy the new implementation across all their servers. Because overloading is not supported in CORBA multiple inheritance has limited applicability. Moreover, applications may need to expose the same IDL interface multiple times to allow developers to either provide multiple implementations or multiple instances of the service through a single access point. Unfortunately, multiple

inheritance cannot expose the same interface more than once and, alone, it cannot determine which interface should be exported to clients [Henning et Vinoski, 1999].

CORBA *Component Implementation Framework* (CIF) is a framework for constructing component implementations. CIF uses CIDL source files to generate skeletons that contain implementations of various mandatory behaviors of components. Mapping to a programming language is performed in two steps: mapping from CIDL to IDL and then mapping from IDL to a chosen programming language. Implemented and compiled component is packed along with a generated component descriptor file and default properties file into a single Zip file and deployed on a server.

There are two possible ways how components may be instantiated and interconnected. The first one is statically through an *assembly descriptor* which is an XML file providing necessary information about instantiating and interconnecting components. The second way is dynamically either from a source code using *Object Request Broker - ORB* services or using a *CORBA scripting tool*. For this purpose CCM provides a variety of introspection services to determine all provisions and requirements of a component.

OpenCORBA

OpenCORBA [Ledoux, 1999] is a reflective implementation of CORBA in NeoClasstalk [Rivard, 1996], which in turn is an extension of Smalltalk with a metaobject protocol that allows the dynamic replacement of the class of an object and, importantly, the meta-class of a class. The reflective features of OpenCORBA are thus based on the idea of modifying the behaviour of a CORBA service by replacing the meta-class of the class defining that service. Two aspects of CORBA are reified and subject to this mechanism.

First, it allows the dynamic adaptation of the behavior of remote invocations by applying the above idea to the classes of proxies (stubs) and server templates (skeletons). By default, these classes are generated (according to the standard IDL to Smalltalk mapping) as instances of the meta-classes `ProxyRemote` and `TypeChecking`, respectively, which implement the standard CORBA behavior. Replacing these meta-classes with custom ones therefore allows remote invocations with different non-functional properties. For instance, it is possible to implement a replication strategy at client side, or to introduce an optimized form of type checking of server invocations.

The other aspect that can be subject to meta-class change is the creation of meta-information elements in the Interface Repository, allowing the adaptation of the strategy for validating the integrity of such elements.

The OpenCORBA approach allows for arbitrary customisations based on behavioural reflection. The facilities provided are an equivalent of request interceptors in CORBA, regarding the customisation of request behavior.

OpenCORBA, however, provides dynamic customization, whereas CORBA interceptors are static. In addition, such customization can be made on a per-interface type basis, whereas CORBA interceptors apply to all requests on an ORB instance.

In general, the OpenCORBA approach is limited to the aspects described above (although the overall approach could apply more generically), thus not allowing, for example, the internal ORB

mechanisms to be customized or adapted.

Another limitation derives from the use of the meta-class approach, meaning that the scope of adaptation is a class, thus affecting all of its instances. For instance, as stub and skeleton classes are generated for each particular interface type, all the running instances of an interface type are affected by a meta-class change (as all of them share the same proxy and skeleton classes).

Similarly to CCM, the structural and behavioral inheritance mechanism is not present in the component level of abstraction.

Meta-ORB

[Costa *et al.*, 2006] proposes the design-time use of models to generate middleware configurations and run-time use of the same models to represent middleware components which are causally connected with their models. The models are then maintained by the reflective meta-objects for the purposes of dynamic adaptation. The MetaORB meta-model is an extension of the CCM meta-model, in a way that it allows backward compatibility with the standard. MetaORB provides the meta-information management with a principled reflective meta-level. This has the benefit of unifying the use of meta-information in the system (*e.g.* preventing that different meta-object implementations use different meta-level representations), as well as providing a basis to closely integrate the configuration and adaptation features of the platform.

Meta-information describes the structure and semantics of entities in a computational system. This description is used for static configuration of the middleware (by instantiating its components at load time) and for its dynamic adaptation (via run-time component-based reconfiguration).

In the MetaORB architecture, meta-information is specified in a model, according to an explicit meta-model. This explicit meta-model represents the platform's type system and is maintained in a repository that can be used for the definition, storage and retrieval of models that represent specialized configurations of the middleware and its applications. Once the definition of an entity (a component and its interfaces, for instance) is obtained from the type repository, it may be used to build a run-time model of the entity, allowing its dynamic instantiation by specialized factories and, if necessary, the construction of its reflective self-representation, used for dynamic introspection and reconfiguration.

MetaORB reflection is based on per-object meta-objects, enabling to isolate the effects of reflection. The MetaORB meta-model reifies the first-class constructs: interfaces (access points to the services provided by a component), components (represent the units of functionality) and binding objects (equivalent to distributed components, whose internal components can be deployed across the network). The corresponding meta-model elements (meta-types) represent both the type and template aspects of such constructs, meaning that the meta-model provides a basis for the functions of type and configuration management. In short, base-level objects (components) are represented as multiple meta-model elements, as it is shown in Figure 2.8

Similarly to CCM, the structural and behavioral inheritance mechanism is not present in the component level of abstraction.

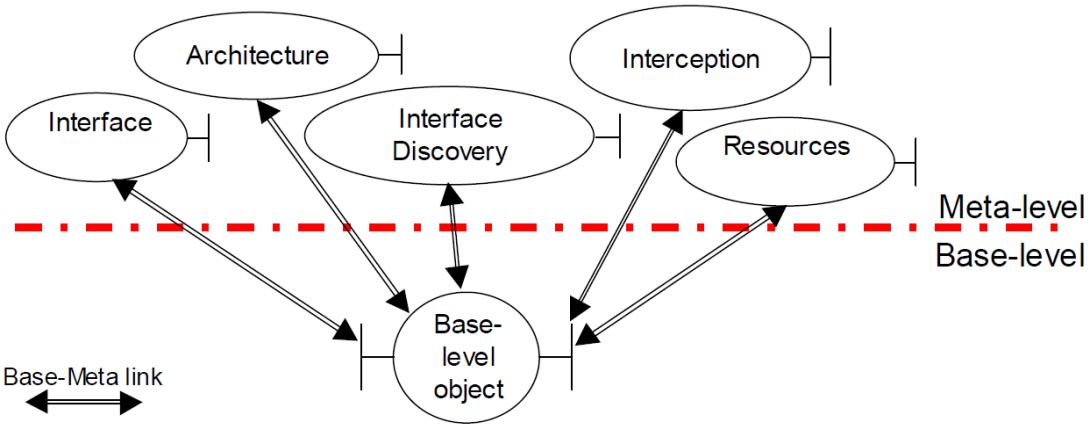


Figure 2.8 : Reifying a base-level object according to multiple meta-space.

DynamicTAO

DynamicTAO [Kon *et al.*, 2000] is a CORBA compliant reflective ORB, which makes explicit the architectural structure of a system in a causally connected way. Component configurators keep the consistency of dependencies as new components are added or removed from the system. Reflection capabilities are limited to coarse-grained components, without possibility to control more detailed structures of the model.

DynamicTAO is based on the ability to dynamically reconfigure the internal strategies of the ORB, by plugging and unplugging strategy implementations on existing components. It defines specialized component configurator classes, in order to provide the dynamic management of different kinds of entities, such as particular ORB instances or ORB domains. Each of these configurators defines a number of hooks for the installation of strategies, depending on the kind of the component it configures. The interfaces of these configurators constitute the DynamicTAO Meta Object Protocol (MOP), with facilities for loading and installing new strategies, and for inspecting the state and structure of the reified components.

Customization and dynamic adaptation of ORB components in DynamicTAO is based on strategies with emphasis on environments with very limited resources, such as handheld computers. The architecture is based on a configurable ORB skeleton, which defines abstract components that represent customization slots for the several ORB services. It also allows the flexible selection of concrete components for each abstract component.

Similarly to CCM, the structural and behavioral inheritance mechanism is not present in the component level of abstraction.

JavaBeans

The *Javabean* component model[8] was developed by Sun Microsystems[9] in 1996 around the Java programming language. A *Javabean* is "*a reusable software component that can be manipulated in a graphical development environment*" [Hamilton, 1997]. It should be noted that all *Javabeans* are not necessarily graphical components (called *widgets*) such as buttons, menu bars, etc.. Even though this model is particularly well suited for building graphical user interfaces, its usage can be much wider.

A *Javabean* is an instance of a Java class that has attributes, methods, *properties* and can emit and receive *events* (*see* Figure 2.9) . Attributes and methods are standard concepts in Java, unlike properties that are "units" of configuration that affect the appearance or behavior of a *Javabean*. A property has a name, a type and a value that is read and/or written via *Javabean* methods which conform to the naming conventions of Java. Events are Java objects that are exchanged when the components are connected. There are a multitude of predefined events and it is possible to define new ones. In most, events are usually related to changes of properties values.

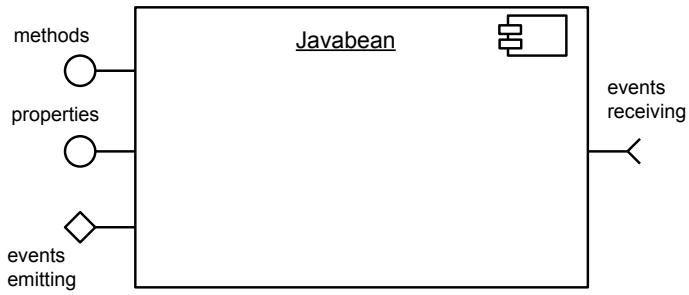


Figure 2.9 : Structure of a *Javabean* component

Javabeans connection is based on notifications and event listeners. Each *Javabean* listen to a set of events. It can react to these events. It can also notifies its listeners with events that it produces. In summary, two independently developed *Javabeans* can be connected (without changing their code) if the former is able to listen and handle events that the latter emits. Connections are defined within an application code by registering a bean as an event listener of another bean.

Javabeans use the Java inheritance mechanism to reuse structure and behavior definition of components.

A *Javabean* also has a standard introspection mechanism accessible through its interface `BeanInfo` which offers information about the bean (its properties, the types of emitted or handled events and its methods). Development environments offered for *Javabeans*, as the *Bean Development Kit* (BDK) or Netbeans, use this mechanism to provide a graphical representation of *Javabeans*.

A bean shelf (library) comes in the form of an archive (jar) containing the bean implementation (compiled Java files) and resource files (configuration files, images, etc..). This archive can then be

[8]The meaning of the word *bean*, in the case of *Javabean*, is "grain". So a *Javabean* is a coffee bean.

[9]On January 27, 2010, Sun was acquired by Oracle Corporation for $7.4 billion, based on an agreement signed on April 20, 2009. The following month, Sun Microsystems, Inc. was merged with Oracle USA, Inc. to become Oracle America, Inc.

easily deployed. The execution of a *Javabean* is supported by the Java virtual machine that acts as a container. This makes the *Javabeans* mobile since JVMs are available for most hardware platforms.

Enterprise Java Beans (EJBs)

EJB (*Enterprise Java Beans*) is a component model developed by Oracle with actual Version 3.0 [Oracle, 2012 ; Monson-Haefel, 1999]. EJB is primarily used for a client-server model of distributed computing, where clients connect to a server in order to access services provided by the server with an emphasis to access relational database.

EJB specification introduces three kinds of components: *Entity classes*, *Session beans* and *Message-driven beans*. In the case when beans export remote interfaces, communication between client and beans and also between beans is performed using RMI (*Remote Method Invocation*) which is a Java implementation of RPC (*Remote Procedure Call*).

The main purpose of entity classes is to access remotely over network data stored in a database or another permanent storage. Each entity class represents an object view on one record from a database, and is therefore identified by a primary key. Due to permanent storage background, entity classes are statefull. Entity classes may be shared between multiple users, that may use a primary keys to access a concrete class. Invocations are performed synchronously.

Session beans are not permanent and have no primary key since are not backed by a database or other form of permanent storage. Session beans are not shareable in general. However persistency and shareability may be achieved by explicit access to a database and use of beans handle. Invocations of session beans are synchronous. Session beans may be statefull or stateless. A statefull bean maintains its state across various method calls. It is intended to be used by one remote client. On the other side stateless bean does not hold its state and may be pooled and used by various remote clients in an instant.

Message-driven beans do not represent any data directly, however they may access any shared data in an underlying database. Message-driven beans are executed when a message from a client is received on a server, so their invocation is asynchronous.

Beans expose two kinds of interfaces:

- A *remote interface* represents provisions of a bean. It provides an access point for a client to access methods of a bean and must be implemented by a developer of a bean.
- A *home interface* provides methods for beans configuration during deployment, access to beans' metadata, and managing the lifecycle of its instances (creation, destruction, research case of persistence, etc. .) A home interface is automatically provided by an EJB *container*.

Both kinds of EJB interfaces are provided interfaces. EJB does not support required interfaces of beans. Only requirements in form of co-operating EJB may be specified within a Deployment descriptor. A reference to related EJB must be however obtained programmatically within a code of a bean and thus an application architecture is hidden.

An application server embed an EJB container for running beans. Beans are deployed together with a deployment descriptor which is a single XML file. An EJB server usually provides various services similar to Corba Object Services: naming and trading service, transaction service and others.

The interface `EJBHome` supports introspection with a method to retrieve the `EJBMetaData` object to obtain information about the enterprise Bean. The `EJBMetaData` object implements the `javax.ejb.EJBMetaData` interface which defines methods for obtaining the class of the bean's remote interface, home interface, bean type (entity, statefull or stateless session), and the primary keys type (entity only). A reference to the bean's EJB home can also be obtained. Once a client application has a reference to bean's remote and home interface classes, normal Java reflection can be used to introspect the methods available for the client. The `EJBMetaData` is designed to be used by IDEs and other builder tools that may need generic methods for obtaining information about a bean at run-time.

Similarly to *Javabeans*, EJBs use the Java inheritance mechanism to reuse structure and behavior definition of components.

OSGi

OSGi [OSGi Alliance, 2012] was originally designed for embedded systems, but later has been used as a general-purpose component model in different domains. The model tries to provide the standard implementation of dynamic modules for the Java platform. The OSGi specification therefore defines a framework for managing the life-cycle of a set of components stored in a module concept called *Bundle*. The two reference implementations of OSGi are Apache Felix[10] and Eclipse Equinox[11] projects.

Applications or components (coming in the form of bundles for deployment) can be (optionally remotely) installed, started, stopped, updated, and uninstalled without requiring a reboot. Application life cycle management (start, stop, install, etc.) is done via APIs that allow for remote downloading of management policies. The service registry allows bundles to detect the addition of new services, or the removal of services, and adapt accordingly.

A *Bundle* means a specific JAR of Java platforms required for deployment. It can define additional meta-information in the form of a manifest file and declare dependencies to other *Bundles* or fragments of *Bundles* exploiting the notion of Java packages. The dependencies have the folowing atributes: name, interface (Class of the service registered in the framework) and cardinality. The cardinality is expressed with the following syntax: `cardinality ::= optionality '..' multiplicity`, for example `0..n` means optional and multiple.

In its primary form the OSGi framework therefore introduces modularity to Java. Modules (bundles) are easy to deploy since the granularity of dependencies is either a JAR package or a Java class. These are accompanied by a Java code to be executed (at start and stop). The code is defined in specific classes named `Activator` and representing the internal code of a component. The framework also defines a concept of internal service whose registration is dynamic in a central registry of the platform.

[10] http://felix.apache.org/
[11] http://www.eclipse.org/equinox/

In short we can say that components are less formally defined than services.

A service is any object that is registered in the OSGi Service Registry and can be looked up using its interface name(s). The only prerequisite is that a service should implement some interface.

In contrast, a component tends to be an object whose life-cycle is managed, usually by a component framework such as Blueprint[12] or iPOJO[13]. A component may be started and stopped; this would be considered an "active" component. A component that does not need to be started or stopped is called "passive". A component may publish itself as an OSGi service. A component may bind to or consume OSGi services. For example a developer can say that his/her component "depends on" a particular service, in which case the component will only be created and activated when that service is available – and also it will be destroyed when the service becomes unavailable.

SCA standard and its implementation FraSCAti

The standard Service Component Architecture (SCA) [OASIS, 2013] proposed by OSOA (Open Service Oriented Architecture) as the result of a desire to unify component-based and service-oriented architectures. Several companies such as IBM, Oracle or SAP have proposed a model to express both: the notion of service and also the notion of software components and their assemblies to model a full architecture. All interconnections between components thus follow the service paradigm which serves to type components contracts. The best known implementations of the standard are Apache project Tuscany[14] or INRIA's project FraSCAti [Seinturier *et al.*, 2012].

The SCA model is defined around four main principles: programming language independence, IDL (Interface Description Language) independence, communication protocols independence, SCA non-functional properties independence. It advocates the principles of service composition and reuse: a system can be composed of new services specifically tailored for the intended application, as well as of components extracted from existing systems and/or applications. SCA provides support for a wide spectrum of programming languages and frameworks (*e.g.* BPEL, PHP, Java) and diverse communication mechanisms (*e.g.* Remote Procedure Call, Web services).

The model defines systems in terms of service *components* and *composites*. The former implement and use services; the latter describe the assembly of components from the point of view of its function. This includes connections between components/services and the references the system offers for its use. Other concepts have been proposed in SCA, like *wires* that connect services (provided ports) to *references* (required ports), *interfaces* that provide a description of both services and references, and, at last, *binding*, which introduces an access mechanism used by services and references. A visual representation of these concepts can be seen in Figure 2.10

The use of SCA within programming languages such as Java is very similar to Fractal, and is performed by introducing annotations into the source code for SCA elements. The standard is organized around a set of four sets of specifications: assembly language, component implementations, bindings, and policies.

[12] http://aries.apache.org/modules/blueprint.html
[13] http://felix.apache.org/documentation/subprojects/apache-felix-ipojo.html
[14] http://tuscany.apache.org/

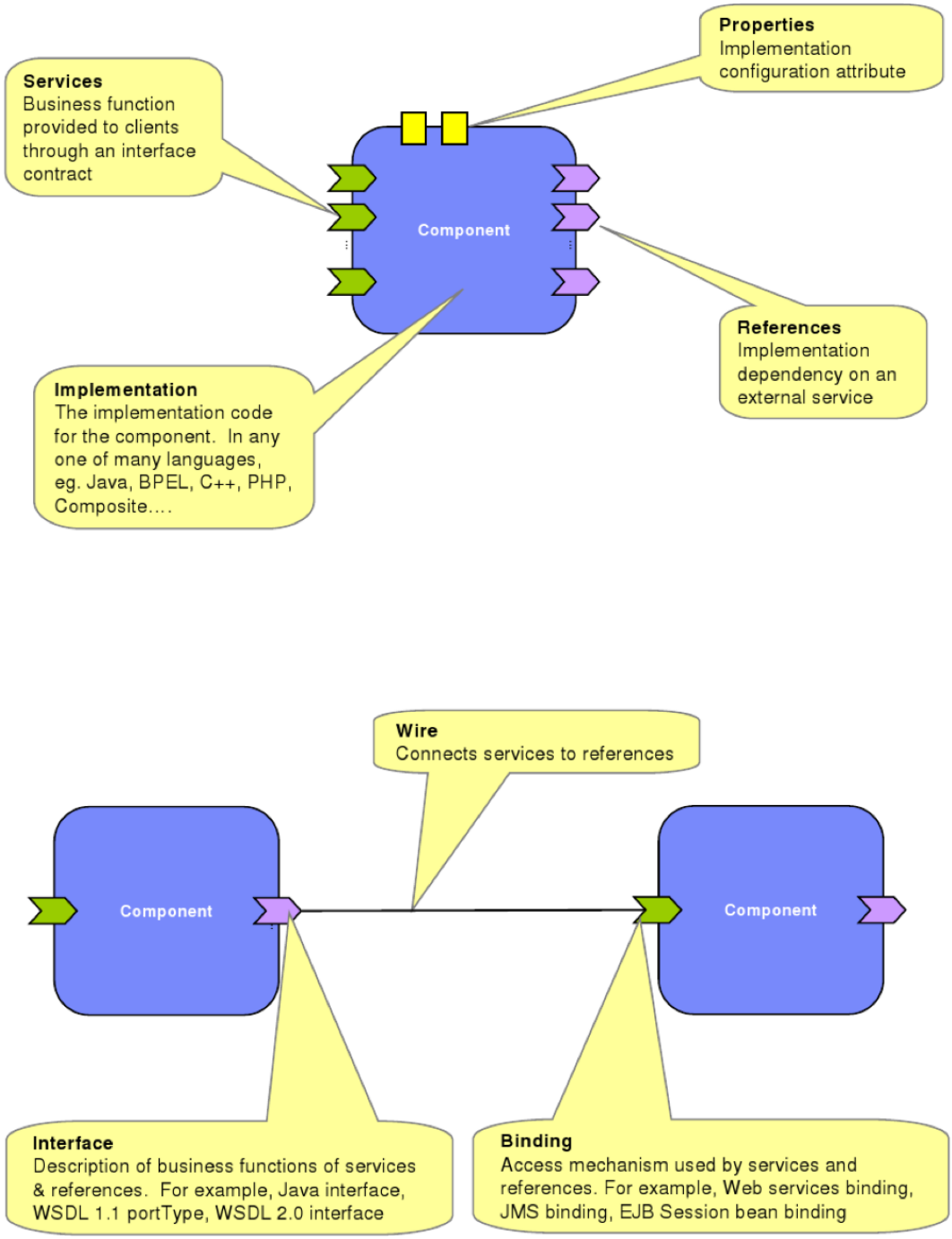


Figure 2.10 : Visual Representation of SCA Concepts

FraSCAti is a model for the development of highly configurable SCA solutions. The main contribution of FraSCAti is to address the above issues of configurability and manageability in a systematic fashion, both at the business (application components) and at the platform (non-functional services, communication protocols, etc.) levels. This is achieved through an extension of the SCA component model with reflective capabilities, and the use of this component model to implement both business-level service components conforming to the SCA specification and the FraSCAti platform itself.

Technically, FraSCAti is built on top of the Fractal component model [Bruneton *et al.*, 2006]. In fact, FraSCAti is an advanced example of using Fractal controllers. It provides the following six controllers [Seinturier *et al.*, 2012], each implementing a particular facet of the execution policy of an SCA component:

Wiring Controller providing the ability, for each component, to query the list of existing wires (`lookupFc`), to create new wires (`bindFc`), to remove wires (`unbindFc`), and to retrieve the list of all existing wires (`listFc`). These operations can be performed at run-time.

Instance Controller which creates component instances according to one of the four SCA modes. The `getFcInstance` method provided by this controller returns the component instance associated with the currently running thread.

Property Controller enabling to attach a property to a component (`putFcValue`) and retrieving its value (`getFcValue`).

Hierarchy Controller implementing the hierarchical design of SCA components, where a component is either *primitive* or *composite*. Composite components contain sub-components that are themselves either primitive or composite. The management of this hierarchy is performed by the hierarchy controller, which provides methods for adding/querying/removing the sub-components of a composite.

Lifecycle Controller dealing with multithreaded applications where the reconfiguration operations cannot be performed in an uncontrolled way. For example, modifying a wire while there is a client request under processing. The life-cycle controller ensures that reconfiguration operations are performed safely.

Intent Controller managing the non-functional services attached to an SCA component.

The resulting collaboration scheme between controllers is captured in a software architecture which is illustrated in Figure 2.11.

Compared to the SCA assembly language that only allows the description of the initial configuration of an application, FraSCAti makes this configuration accessible and modifiable while the application is being executed. The following component elements can be changed at run-time: wires, properties, and hierarchies. By providing a run-time API, the platform enables the dynamic introspection and modification of an SCA application. This feature is of particular importance for designing and implementing agile SCA applications, such as context-aware applications.

FraSCAti takes over the reuse capabilities introduced by Fractal's inheritance.

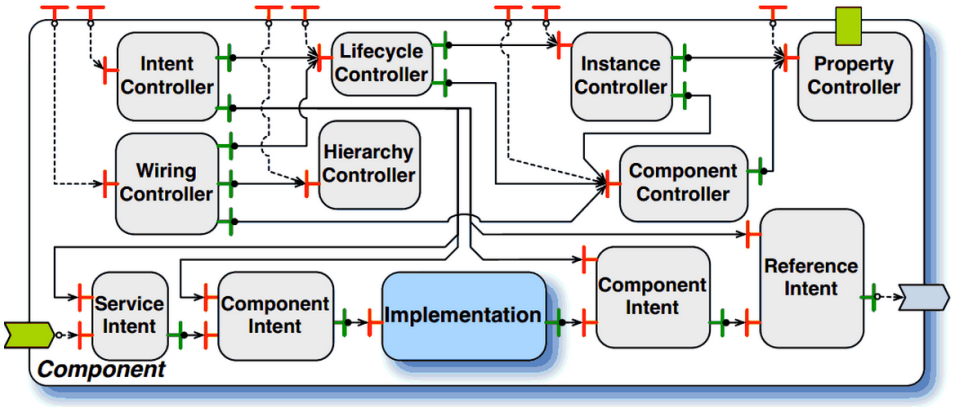


Figure 2.11 : FraSCAti controllers, each implementing a particular facet of the execution policy of an SCA component

2.2.3 Generative family

The *generative family* uses high-level abstraction design models as conceptual tools for managing the complexity of large software systems. These models, usually specified in ADLs, describe the high-level organization of a software system as a collection of components, connections between the components, and constraints on how the components interact. A system's architecture provides a model of the system that suppresses implementation detail, allowing the architect to concentrate on the analysis and decisions that are most crucial to structuring the system to satisfy its requirements. The intent of these models is to communicate to an entire engineering team part of the global knowledge needed to develop and evolve each component of the system. They also aid in the specification and analysis of high-level designs. For example, an architectural model can be analyzed to prove that the design invariants described by architectural constraints are satisfied.

Once the architecture design stage of development cycle is finished, the generative strategy takes the formal description of a designed architecture and generates code skeletons using a generator specific for an implementation language. This implementation-language independence property is another advantage of the approach as the same abstraction layer can generate code for different machines, taking into account the heterogeneity of platforms.

Our study plan focuses on inheritance and reflection, however the static nature of the generative family do not match with reflection very well [Medvidovic et Taylor, 2000]. Reflection or at least introspection capabilities depend on code which is generated. Behavioral inheritance, *i.e.* the ability to reuse the behavior definition of components, is also not supported since the family mainly focuses on structural description of complex software systems, not on the implementation of functionality provided by these systems.

ACME

ACME [Garlan *et al.*, 1997] was developed as a joint effort of several architectural research groups, ACME is intended to serve as a least- common-denominator interchange language for architectural descriptions. It defines a basic, un-interpreted vocabulary of components, connectors, ports, roles, bindings, and configurations. A system is constituted by components connected by connectors; the ports are end-points of the connectors. These elements may have specifications associated with them through property lists. In ACME properties may be specified in any language. The semantics of the properties and, even, of the overall architectural specification are supplied by these auxiliary languages. The goal is that a specification written in one language, say UniCon, could share a common architectural structure with a specification in another language, say Rapide, and thus the architect will be able to take advantage of the descriptive and analytic tools of multiple ADLs or formalisms.

ACME is built on a core ontology of seven types of entities for architectural representation: components, connectors, systems, ports, roles, representations, and rep-maps. Among the seven types, the most basic elements of architectural description are components, connectors, and systems.

- *Components* represent the primary computational elements and data stores of a system. Intuitively, they correspond to the boxes in box-and-line descriptions of software architectures.
- *Connectors* represent interactions among components. Computationally speaking, connectors mediate the communication and coordination activities among components.
- *Systems* represent configurations of components and connectors.

Components' interfaces define points of interaction between the component and its environment. Each port identifies a point of interaction between a component and its environment. A component may provide multiple interfaces by using different kinds of ports. A port can represent an interface as simple as a single procedure signature, or more complex interfaces, such as a collection of procedure calls that must be invoked in a certain specified order, or an event multi-cast interface point. Connectors also have interfaces that are defined by a set of roles. Each role of a connector defines a participant of the interaction represented by the connector. Binary connectors have two roles such as the caller and callee roles of an RPC connector, the reading and writing roles of a pipe, or the sender and receiver roles of a message passing connector. Other kinds of connectors may have more than two roles. For example an event broadcast connector might have a single event-announcer role and an arbitrary number of event-receiver roles.

ACME supports the hierarchical description of architectures. Specifically, any component or connector can be represented by one or more detailed, lower-level descriptions. Each such description is called a representation in ACME. The use of multiple representations allows ACME to encode multiple views of architectural entities. It also supports the description of encapsulation boundaries, as well as multiple refinement levels.

When a component or connector has an architectural representation there must be some way to indicate the correspondence between the internal system representation and the external interface of the component or connector that is being represented. A *rep-map* defines this correspondence.

ACME has focused on providing a common skeleton through which the benefits of other languages and tools can be combined. Thus, ACME is consistent with the ability to describe and analyze software architectures, but it does not, in itself, provide a sufficient basis. It is only through mappings to other languages that ACME descriptions can be interpreted and analyzed. One peculiar thing about ACME is that its representation can vary depending on the underlying model. For instance, a UNIX pipeline could be modeled in a syntax similar to C as we show in Listing 2.3.

```
Component pipe =
{
   Port in;
   Port out;
   Property implementation : String = "while (!in.eof) {in.read; compute; out.write; }";
}
```

LISTING 2.3 : A component modeling a UNIX pipe in ACME

ACME partially supports structural inheritance with the notion of sub-typing via the extends feature.

Aesop

The Software Architecture Design Environment Generator (AESOP) [Garlan *et al.*, 1994] is a set of tools designed to develop a system model which provides a vocabulary for architectural description through an object-oriented framework of types. It is based on the UNIX environment; it has pipe and filter style extensions in order to model those features. However, it has a generic kernel, suitable for all environments. The language does not provide a plain text description of the model; all modeling is done in the graphic editor of the tool.

An architectural configuration is represented as an interconnected collection of object instances. The vocabulary of an architectural style is described by defining sub-types of the basic architectural types: Component, Connector, Port, Role, Configuration, and Binding. By default, an architectural configuration in AESOP is an un-annotated hierarchical structure of components, connectors, and configurations. A style provides attributes for representing the semantics of individual elements and tools for analyzing and exploiting those specialized representations In also modifies the manipulation methods of the base types to enforce style-specific constraints.

For example, a pipe-filter style would define a “filter” sub-type of component, a “pipe” sub-type of connector, and appropriate port and role sub-types. The “insert port” method of a filter, for example, would require that the inserted port be either a data input or output. Additionally, tools could be introduced into the environment to generate implementations of pipe-filter systems from the architectural representation. Aesop can be used to support the analysis techniques of a particular formal method or some other ADL.

AESOP provides component sub-typing support. Aesop enforces behavior-preserving sub-typing to create sub-styles of a given architectural style. An AESOP subclass must provide strict sub-typing behavior for operations that succeed, but may also introduce additional sources of failure with respect to its super-class.

C2

C2 is a component- and message-based style designed to support the particular needs of applications that have a graphical user interfaces, with the potential for supporting other types of applications as well. The style supports a paradigm in which UI components, such as dialogs, structured graphics models (of various levels of abstraction), and constraint managers, can more readily be reused. A variety of other goals are potentially facilitated as well. These goals include the ability to compose systems in which: components may be written in different programming languages, components may be running in a distributed, heterogeneous environment without shared address spaces, architectures may be changed dynamically, multiple users may be interacting with the system, multiple toolkits may be employed, multiple dialogs may be active (and described in different formalisms), and multiple media types may be involved.

The C2 style can be informally summarized as a network of concurrent components hooked together by connectors, *i.e.*, message routing devices. The top of a component may be connected to the bottom of a single connector and the bottom of a component may be connected to the top of a single connector. There is no bound on the number of components or connectors that may be attached to a connector (*see* Figure 2.12).

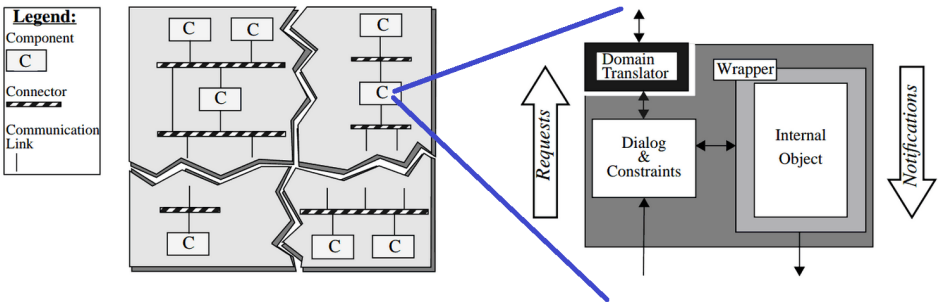


Figure 2.12 : A sample C2 architecture and a detail of the internal architecture of a C2 component. Jagged lines represent the parts of the architecture not shown.

Each component has a top and bottom domain. The top domain specifies the set of notifications to which a component responds, and the set of requests it emits up an architecture. The bottom domain specifies the set of notifications that this component emits down an architecture and the set of requests to which it responds. All communication between components is solely achieved by exchanging messages. Message-based communication is extensively used in distributed environments for which this architectural style is suited. Central to the architectural style is a principle of limited visibility or substrate independence: a component within the hierarchy can only be aware of components "above" it and is completely unaware of components which reside "beneath" it. Substrate independence has a clear potential for fostering substitutability and reusability of components across architectures. To eliminate a component's dependence on its "superstrate," *i.e.*, the components above it, the C2 style introduces the notion of event translation: a transformation of the requests issued by a component into the specific form understood by the recipient of the request, as well as the transformation of notifications received by a component into a form it understands. The C2 design environment is, among other things, intended to provide support for accomplishing this

task.

The internal architecture of a C2 component shown in Figure 2.12 is targeted to the user interface domain. While issues concerning composition of an architecture are independent of a component's internal structure, for purposes of exposition below, this internal architecture is assumed.

C2 supports multiple sub-typing relationships among components: name, interface, behavior, and implementation. C2 allows creation of sub-types of such a component by sub-typing from any or all of the internal blocks. Different combinations of these relationships are specified using the keywords *and* and *not* as we illustrate in the following code snippet:

```
component Well_1 is subtype Matrix (beh)
component Well_2 is subtype Matrix (beh \and \not int)
```

C2 also supports conformance checking mechanisms. It even allows sub-typing from several types, potentially using different sub-typing mechanisms due to multiple conformance mechanisms.

C2 connectors have context reflective interfaces. Each C2 connector is capable of supporting arbitrary addition, removal, and reconnection of any number of C2 components.

Darwin

Darwin [Magee *et al.*, 1995] is an architectural description language developed by Magee and Kramer. It describes a component type by an interface consisting of a collection of services that are either provided (*i.e.* declared by that component) or required (*i.e.*. expected to occur in the environment). Configurations are developed by component instantiation declarations and bindings between required and provided services. Darwin supports the description of dynamically reconfiguring architectures through two constructs — lazy instantiation and explicit dynamic constructions. Using lazy instantiation, a logically infinite configuration is described and components are instantiated only as the services they provide are used by other components. Explicitly dynamic structure is provided through the use of imperative configuration constructs. In effect, the configuration declaration becomes a program that is executed at run-time, rather than a static declaration of structure.

In a Darwin-generated implementation, each primitive (non-hierarchical) component is assumed to be implemented in some programming language, and platform specific glue code is generated for each service type. The elaboration algorithm acts, essentially, as a name-server that provides the location of provided services to any executing components.

The model does not provide any means of describing the properties of either a component or its services. Component implementations are uninterpreted black boxes, while the collection of service types is a platform-dependent collection whose semantics is also uninterpreted in the Darwin framework.

Darwin supports hierarchical composition. A component might be composed of other components or of primitive components which represent built-in features of the language. Darwin also supports the structure of parallel programs and modeling of network topologies.

Darwin's support for architectural style is limited to the description of parameterized configurations. For example, a Darwin pipeline description is shown in Listing 2.4. This description indicates

that a pipeline is a linear sequence of filters, where each filter's output is bound to the input of the next filter in line. Using parameterization to describe families of systems means that only systems that can be described constructively can be effectively characterized. That is, in order to delineate membership in an architectural style it is essentially necessary to construct an algorithm that can construct exactly those members of the style.

```
component pipeline (int n) {
   provide input;
   require output;

   array F[n]:filter;
   forall k:0..n-1 {
      inst F[k];
      bind F[k].output == output;
      when k<n-1 bind
         F[k].next == F[k+1].prev;
   }
   bind
      input == F[0].prev;
      F[n-1].next == output;
}
```

LISTING 2.4 : A pipeline component description in Darwin

Components hide their behavior behind well-defined interfaces and programs are constructed by creating instances of component types and binding their interfaces together. These compositions are considered as types as well, which leads to a hierarchical composition. Interfaces in Darwin can be parameterized and derived by inheritance from one or more base interface types. Also, component types can be defined explicitly or fully or partially typed from an existing component type (a partial component declaration).

Rapide

Rapide [Luckham et Vera, 1995 ; Luckham *et al.*, 1995a] is a set constituted by a type language, a definition language, a constraint language, and an executable programming language. The type language is intended to provide interfaces for the definition language, which defines the architecture. The constraint language defines requirements for timing and other pattern events. The executable language is concurrent and reactive. Its main purpose is to construct behavior of components and connections between components.

The language is based on modeling computations and interactions as partially ordered event sets (or "posets"). Rapide defines component types (called *interfaces*) in terms of a collection of communication events, which are either observed (Rapide calls these "extern actions") or initiated ("public actions"). Rapide interfaces define the computational behavior of a component by relating the observation of extern actions to the initiation of public actions. Interaction between components is described for a particular configuration in one of two ways: (i) events can be "connected", in which case they are aliased to the same event. (ii) an architecture can declare "constraints" which specify causal relationships between events on different components. Given a constraint, the initiation of

one event will result in the generation of another event following it in all event orderings, but they are not considered as the same event.

Rapide's connections define an asymmetrical, primitive relation between two components. The type language of actions permits the definition, essentially, of function calls between components. Events have parameters and possibly return values. As such, they are not adequate for the introduction of new interaction types and do not support symmetric interaction patterns.

Rapide's constraints might appear to provide better support for an explicit connector mechanism, since related constraints can generate quite complex interactions between components. However, these constraints can only be declared at the global configuration level and therefore do not permit the localization of analysis of interactions. Further, because the language does not permit them to be explicitly bundled as connectors, complex interaction patterns (sets of constraints) can not effectively be reused in multiple contexts.

Rapide permits a form of consistency checking and analysis through architectural simulation. In essence, the architecture is simulated, generating a partially ordered set of events that is compatible with the interface, behavior, and constraint specifications. Because the generated set explicitly defines causal relations between events rather than simply providing one possible sequence of the events, simulation is useful for detecting execution alternatives such as race conditions.

However, a given simulation execution will provide only one possible poset, rather than all permitted posets. This means that alternatives due to non-determinism in a behavior specification are not captured by architectural simulation.

Architectures in Rapide can be filled in with implementations in an executable sub-language or in languages such as C++ or Ada. The Rapide system includes a tool that dynamically monitors the execution of a program, checking for communication integrity violations. Communication integrity could be enforced statically if system implementors follow style guidelines, such as never sharing mutable data between components. However, the guideline forbidding shared data prohibits many useful programs, and the guidelines are not enforced automatically.

The type language allows deriving new interface type definitions by inheritance from existing ones, including the capability of dynamic substitution of sub-types for super-types. However, at the higher levels (Architecture, ...), inheritance is not supported.

UniCon

UniCon [Shaw *et al.*, 1995] provides a tool for constructing executable configurations based on component types, implementations and "connection experts" that support particular connector types. UniCon supports explicit, symmetrical and asymmetrical connectors. That is, an architectural configuration contains connector declarations that logically define an interaction. Each connector has a collection of roles that define which participants are expected in a given interaction. Component interfaces, rather than providing or requiring services, are defined by players which have a type (indicating the nature of the interaction expected) and a set of properties (providing details of the component's interaction at that interface). At the configuration step, players on components are associated with roles on connectors.

While UniCon's model of explicit connectors seems to provide promise for architectural style, by creating a place where new interactions might be defined and compositional rules elaborated, UniCon currently provides limited mechanisms for defining new connector types. New types can only be added by hand-implementing new connection experts. This adds to the collection of built-in, atomic types. Every connector is tagged with a type, which is manifested as a type construct in the written notation and as the choice of an icon in the graphical notation. This type indicates which roles must be satisfied for the connector to operate properly, together with the types of players that are eligible to play the roles. Analysis of architectures in UniCon is limited to those tools supplied with the given connector types (which depend on specific implementations of components and connectors) and there is no way to describe architectural styles.

A component's interface consists of the component's type, specific properties (attributes with values) that specialize the type, and a list of points (players) through which the component can interact with the outside world. Each player is typed and may list properties that further specify the player. Property lists are used to refine the types to subtypes or to specialize a type to a particular use. A component's implementation may either be primitive or composite. A primitive implementation consists of some element outside of UniCon's domain, such as a source file in a given programming language, a data file, or an executable. A composite implementation consists of other components and connectors, composed as described below. A connector's protocol consists of the connector's type, some specific properties that specialize the type, and a list of points (roles) at which the connector can mediate the interaction among components. Each role is typed and optionally lists properties that further specify the role. UniCon supports only built-in connectors, so each connector's implementation is specified as built-in.

Wright

```
Configuration Capitalize
Component UpperCase
Connector Pipe
Instances
   Split : SplitFilter
   Upper : UpperCase
   Merge : MergeFilter
   P1, P2, P3 : Pipe
Attachements
   Split.left as P1.Source
   Upper.Input as P1.Sink
   Split.Right as P2.Source
   Merge.Right as P2.Sink
   Upper.Output as P3.Source
   Merge.Left asP3.Sink
End Capitalize
```

LISTING 2.5 : A component modeling a filter in Wright

WRIGHT is designed to support the formal description of the architectural structure of software systems. In order to do so, it permits the description of both architectural styles, or families of sys-

tems, and architectural instances, or individual systems.

WRIGHT is built upon the following abstractions: components, connectors, and configurations. The language provides explicit notations for each of these elements, formalizing the general notions of component as computation and connector as pattern of interaction.

A WRIGHT specification describes a component interface as a collection of ports, or logical interaction points. Each port is defined in terms of a protocol written in a subset of CSP [Hoare, 1978]. These ports factor the expectations and promises of the component into the points of interaction through which the component will interact with its environment. The component may optionally further specify how the interactions on its ports are combined into a computation.

The configurations can be divided into instances (a type if a specification of a component), attachments (describes the topology of the system), and hierarchy (a component may hold other components). In Listing 2.5 we give an example of a filter routine.

Components have two important parts — an interface and a computation. The interface consists of several ports. Each port represents an interaction. The computation provides a more complete description of what is done.

Connectors are considered as composition patterns among components. A collection of interface instances combined via connectors is called a configuration.

No inheritance notion is present in the model.

UML

UML (*Unified Modeling Language*) [OMG, 2011b] is a standard graphical language defined by the OMG for modeling systems. In its version 1.x, this language already included the notion of components, considered modular, expandable and interchangeable system units encapsulating the implementation and outlining a set of interfaces. This low level vision components for a modeling language has changed from UML 2.0. The components are now units of abstract structure that represent subparts of a system. They can be modeled from different points of view and refined throughout the development cycle.

UML 2.0 has introduced concepts and mechanisms inspired by ADLs for describing systems in terms of interconnected components in the *Internal Structures framework*. The framework is used to capture the internal structure of a component (and can be also used for hierarchical components). The meta-class `StructuredClassifier` allows to decompose the functionality of a Classifier into several parts. A part is a `Property` of the owning `StructuredClassifier` referenced via the parts association. Technically, the type attribute of a part specifies the type of the classifier that will be instantiated within an instance of the owning structured classifier. Further, parts may be interconnected via connectors, which correspond to future links to be established among the corresponding instances.

A component has an *external structure* made of *ports* (*see* Figure 2.13). Ports isolate a component from its environment, with the interactions between the *internal structure* and its environment.

Several ports can be defined for a component, which can distinguish different interactions depending on the port through which they are made. Ports enhance the decoupling between a com-

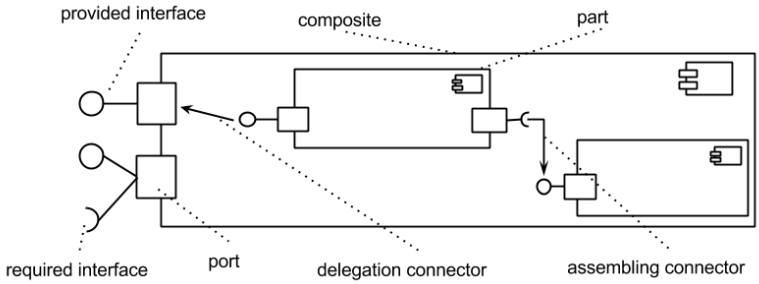


Figure 2.13 : Structure of a UML component

ponent and its environment so it can be reused in another environment that meets the constraints imposed on the port. Constraints associated with a port can be *provided interfaces*, *required interfaces* or *protocols*. A port can be associated with several provided and required interfaces. Interfaces describe static constraints (type and signature operations) that must be implemented by a *classifier* (class or component). Protocols are descriptions of dynamic constraints that can be done using state diagram which we give an example in the following.

Collaboration between two or more components result in *connectors* representing opportunities for communication between multiple instances of components. There are two types of connectors in UML: the *assembly connectors* and *delegation connectors*.

An assembly connector is used to connect components that provide and require consistent services. A connector assembly is defined between a required interface (resp. a required port) and a provided interface (resp. specified port). Delegation connectors are used in the construction of composites, that is to say components with an internal structure comprised of *Parts* (*Property* derived from *TypedElement*), and connectors. Delegation connectors are used to connect external contracts a component interfaces (or ports) to external contracts such to share that redirect requests inbound operations. As for ports, a connector can be described by a protocol and it is then possible to check the validity of a connection by checking the compliance of protocols.

The UML-meta-class `Component` also inherits from the UML-meta-class `Class` (as redefined in the Structured Classes package) and gains the ability to have methods and attributes and to participate in associations and generalizations. Further, besides the associated ports, a `Component` may be also directly associated with a set of provided and required interfaces.

Although UML is a reflective model described in MOF, which is the core sub-set of UML, its reflection capabilities are not usable in run-time.

2.2.4 Component-oriented languages family

Component-oriented languages family states that the most natural way to develop component-based software systems is to use a single programming language that allows doing so in the first place. Such programming languages provide a primitive support for both component definition, and composition (building components by assembling smaller components).

ArchJava

ArchJava [Aldrich *et al.*, 2002] is an extension to the Java Programming Language that allows the unification between the software architecture and its implementation using the same programming language, thereby simplifying the development process. The main purpose of ArchJava is to introduce a programming language to deal with components that guarantees communication integrity between architecture and implementation. A system has communication integrity if components only communicate directly with the components they are connected to in the architecture of component classes. This property ensures that the software system implementation respects the desired system architecture. The presence of a sound type system guarantees communication integrity between an architecture and its implementation, even in the presence of shared objects and run-time architecture configuration.

The three main ingredients of the ArchJava programming language are components, ports and connections. Components are obtained from the instantiation of component classes, and are seen as special objects that communicate with other components in a structured way. Ports represent logical communication channels between one component and the ones it is connected to, making it the only possible way for two components to communicate with each other. Ports declare three different sets of methods: requires, provides and broadcasts. The first set represents methods that are implemented by another component but which are available at the current port (this implies a connection between the component that implements the method and the one that uses it). Provided methods are implemented by the current component, and are made available to other components at the port that is being defined. Broadcast methods are very much like required ones except that they can be connected to more than one implementation, while required ones only allow the connection to a single implementation.

This approach also supports hierarchical software architectures where components have got internal component structures (components connected to each other) to define their functionality. The outer components are called composite components, and the inner ones called sub-components. ArchJava does not present ways for exporting the behavior of internal components to outside components.

An interesting feature of ArchJava is the possibility to create dynamic architectures, *i.e.* architectures that change during the execution of the program, where new components can be dynamically instantiated and connected to each other.

ArchJava's reflection package is an extension of Java reflection package which is mostly read-only, *i.e.* supports introspection, not full reflection. The ArchJava reflection package offers the follwing classes:

- Call - represents a particular run-time required method invocation on a connected port
- Connection (extends Element implements Serializable) - represents a connection between ports at run time
- Connector - represents a custom connector. Methods *typecheck* and *invoke* can be overridden by sub-classes to get custom checking and method invocation semantics. Has a connection as its private attribute.

- Element - represents an element of the parse tree. Used as a way to tell the system what syntactic element caused a type error
- Method (extends Element implements Serializable) - represents a method in a port instance
- Port (extends Element implements Serializable) - represents a port instance
- Type - represents the type of a method argument or result

Component classes can inherit from other component classes, or from Java class `Object`. Component sub-classes inherit methods, ports, and connections from their super-classes. Component sub-classes may also override method definitions and specify new methods and ports. However, component sub-classes may not specify new required methods because this could break sub-type substitutability. ArchJava also supports architectural design with abstract components and ports, which allow an architect to specify and type-check an ArchJava architecture before beginning a program implementation.

Example The example developed in this section is an adapted example inspired by [Aldrich, 2003]. Listing 2.6 shows the code of composite WEBSERVER. In this example, the sub-component `Router` accepts HTTP queries and transmits to a `Worker` who treats. With each new incoming request, the `Router` requests a new `Worker` (see line 29) through its port `request` to process the request. This port `request` is connected to the private port `create` of the composite WEBSERVER (see line 3). In the implementation of the `requestWorker` method of WEBSERVER a new instance of WORKER is created and connected through the port `serve` (see line 11). This dynamic connection is valid because according to the connection pattern in line 4.

```
   public component class WebServer {
2     private final owned Router r = new Router();
      connect r.request, create;
4     connect pattern Router.workers, Worker.serve;

6     public void run() { r.listen(); }

8     private port create {
        provides r.workers requestWorker() {
10          final owned Worker newWorker = new Worker();
            r.workers connection = connect(r.workers, newWorker.serve);
12          return connection;
        }
14    }
   }
16
   public component class Router {
18    public port interface workers {
        group stream;
20      requires void httpRequest(stream InputStream in,stream OutputStream out);
      }
22    public port request {
```

```
      requires this.workers requestWorker();
   }
   public void listen() {
      unique ServerSocket<stream> server = new ServerSocket(80);
      while (true) {
         unique Socket<stream> sock = server.accept();
         this.workers conn = request.requestWorker();
         conn.httpRequest(sock.getInputStream(), sock.getOutputStream());
      }
   }
}

public component class Worker extends Thread {
   public port serve {
      group stream;
      provides void httpRequest(stream InputStream in,stream OutputStream out) {
         this.in = in; this.out = out; start();
      }
   }
   public void run() {
      // gets requested file and sends it on the output stream
   }
}
```

LISTING 2.6 : ArchJava's code of components WEBSERVER, ROUTER and WORKER.

ACOEL

ACOEL [Sreedhar, 2002] is a component-oriented extensional language for creating and plugging components together. The design of ACOEL was motivated by the following component design principles.

- *Pluggable Units.* A component is a unit of abstraction with clearly defined external contracts and the internal implementation should be encapsulated. The external contract should consist of both the services it provides and the requirements it needs when it is plugged or (re-)used in a system.
- *Late and Explicit Composition.* For a component to be composable by a third-party with other components, it must support late or dynamic composition. During the development phase, requirements of a component should only be constrained by some external contract. Then, at run-time, an explicit connection is made with other "compatible" components to achieve late composition.
- *Types for Composition.* Typing essentially restricts the kinds of services (*i.e.* operations or messages) that can be requested from a component.
- *Restricted Inheritance.* In OOP, it is well-known that one cannot achieve both true encapsulation and unrestricted class inheritance with overriding capabilities. In ACOEL, classes (which support inheritance) are second-class citizens, and are not visible to the external clients.

The main construct in ACOEL is component. A component in ACOEL consists of (1) an external contract made of typed input and output ports, and (2) an internal implementation consisting of classes, methods, and data fields. A client can only see the external contract and the internal implementation is completely encapsulated. A component provides services via its input ports, and specifies the services it requires via its output ports. In ACOEL, a connect statement makes an explicit connection between an output port of a component to a "compatible" input port of another component. To each port a class implementing the behavior offered or demanded via the port is attached. Any messages that arrive at an input port are forwarded to the instance of the class that is attached to the input port. The class instance will either process the message or it will delegate to another class instance inside the component.

ACOEL model supports a kind of behavioral inheritance by the *extend* statement which allows to "decorate" inherited ports with mixins, as it is shown in Listing 2.7. This decoration makes it possible to specialize inherited behavior when an inherited service is re-implemented in the decorating mixin. A child cannot access any of the internals (implementation classes, methods) of a parent, except via the input ports of a parent, i.e. `This.<portname>.<servicename>` (composition-like approach). The advantage of this black-box approach is that it preserves encapsulation of parent components.

```
component Foo {
  in I fin <mixin M <: I1>;
  class Cy implements I;
  class Cx = M + Cy;
}
component Goo extends Foo {
  in I fin <mixin M <: I2>
  ...
}
mixin Moo implements I2 { ... }
Goo g = new Goo(){fin<Moo>}
```

LISTING 2.7 : ACOEL mixins

ComponenJ

ComponentJ [Seco *et al.*, 2008] offers a general component model described in the form of a core typed programming language whose first-class values are objects, components, and configurators (*see* Figure 2.14). In this abstract programming model, objects are component instances (*cf.* class instances) which aggregate state and functionality in the standard object-oriented sense. Components are the entities that specify the structure and behavior of objects by means of a combination and adaptation of smaller components and user-defined building blocks. Each component is defined by a network of elements which is specified by a functional-like value, a configurator.

The implementation of services provided by component instances is defined by combination and adaptation of services provided by smaller components. At the component level, ports play an important role as the connection unit between elements. A component declares a set of required ports, which denote abstract implementations of external services, and a set of provided ports which it must

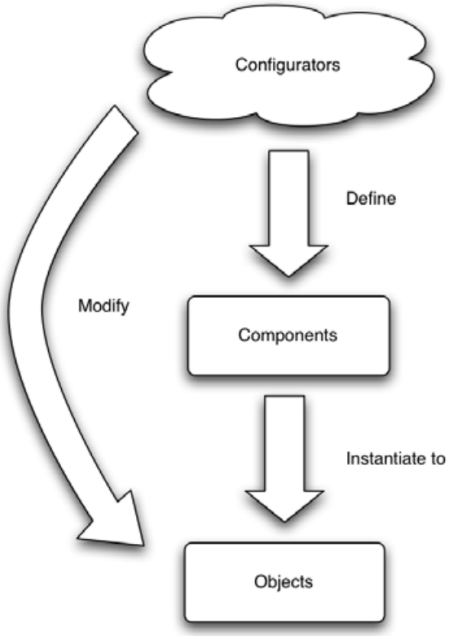


Figure 2.14 : ComponentJ model ingredients and interactions.

implement. In fact, the implementation defined in a component is parameterized in its required ports, which must be satisfied before the component is used to produce objects.

Configurator values denote composition operations which aggregate or connect existing elements in an implementation-independent way, and are uniformly used to produce components or modify the internal structure of objects. Thus, this variety of values and language constructs allows both the expression of dynamic construction of new components (based on run-time decisions) and the unanticipated reconfiguration of component instances.

The ability to express component structures at a high-level of abstraction enable the static verification of structural soundness of components and objects, by means of a type system. In particular, typing configurators with intensional type information, revealing certain aspects of their internal structure, permits type safe composition and reconfiguration actions to be performed on run-time values. Reconfiguration actions are, to some extent, a violation of the encapsulation principle ensured by the type system.

The novelty of this approach when compared with other component models lays in the dynamic construction and run-time modification of the structure and behavior of objects in a statically typed Java-like setting. ComponentJ provides full computational power to build sophisticated and declaratively defined networks of objects while clearly maintaining the definition of architecture and computation separate. The construction of new components and reconfiguration of objects and corresponding soundness properties are defined together in a unique programming language and a single type system. This provides the language with a higher level of expressiveness and statically ensures the absence of run-time errors due to ill-formed component structures.

ComponentJ is an inheritance-free language where authors prefer to avoid inheritance in favor of object composition. Reflection is not explicitly advocated in ComponentJ. It however appears that a running system certainly has a partial representation of itself to allow for dynamic reconfiguration of internal architectures of components as described in [Seco *et al.*, 2008] but it seems to be a localized and ad.hoc capability. The reification process being neither explicited nor generalized.

Clic

CLIC [Bouraqadi et Fabresse, 2009], an extension of SMALLTALK to support full-fledged components. It provides component features such as ports, attributes, or architecture. From the implementation point of view, it fully relied on SMALLTALK reflective capabilities. Thus, from the SMALLTALK virtual machine point of view, CLIC components are objects and their descriptors are extended SMALLTALK classes. Because of this symbiosis between CLIC and SMALLTALK, the use of CLIC allows taking benefit from modularity and reusability of components without sacrificing performance.

A CLIC component has ports. The model is a unidirectional model and thus distinguish provided and required ports. All interactions with a component have to pass through one of its ports. CLIC model allows components to have only one provided port. The idea of a single provided port is based on the observation that developers do not know beforehand, which services will be specified by each required port of client component. Therefore it is hard to split component functionality over multiple ports. CLIC also support explicit architecture description and inheritance. A CLIC component inherits every part of the definition from its parent. A sub-component can override attribute initialization directives and extend the other features (attribute declarations, required ports, architecture ...).

Bichon

Bichon [Xu et Ren, 2010] is a Java-based COL whose design seeks to address the needs of COP and avoid lacks of component-based software design using OOP. Bichon do not extend the Java language directly. It builds on top of Java with component composition instead of inheritance, retaining only the basic data types of the Java language. Composition mechanism avoids problems such as fragile base class caused by inheritance.

In Bichon, components are first-class core language abstractions. The model proposes a class/instance-like approach and clearly distinguish these two concepts. A component instance is a run-time entity while a component describes a template for possibly multiple component instances. In order to achieve separations of static behavior and component run-time behavior, Bichon component model defines two types of interfaces: mixinner and connector. Components are black boxes and all the interactions between them happen only via the mixinner interfaces and the connector interfaces.

By defining these two kinds of interfaces, the authors offer a way to separate run-time behaviors from static behaviors. The principle of separation of concerns is also achieved. At compile time, static mixin between components happened on the mixinner interface. At run-time, components interact and communicate with each other via the connector interface. With this separation, overriding only happens on mixinners, and message dispatch only happens on connectors. All interactions among

components are achieved through these two kinds of interfaces. Bichon do not provide other declarations of message passing.

The model introduces the bi-directional interface supporting bi-directional interactive relationship. Required declarations define view points for the interface on the environment, while provided declarations define viewpoints on the interface for the environment.

When interaction occurs, the interface of components involved must match each other. The type system ensures that only the compatible interface could interact with each other and match with each other.

The model does not reify component related concepts like ports or connection and its reflection capabilities are the ones of Java reflection.

2.3 Comparison

The study plan presented in the end of Section 2.2.1 puts focus on the three following aspects: external contract & architecture description, inheritance for structural and behavioral reuse and reflection capabilities. In this section we present a comparison for each family of component-based approaches regarding these three aspects.

Frameworks family All surveyed models support specification of the external contract of components via declaration of ports. They differ in the terminology and the kinds of information they specify. For example, while SOFA uses terms provided and required interface for ports, CORBA CCM uses terms facets and receptacles, respectively. COM, *Javabeans* does not support specification of requirements via required ports. EJB and OSGi enable users to define requirements (called dependencies) in external files like manifest file (OSGi) or Deployment descriptor (EJB).

Only 4 of 13 models support explicit description of architecture and hierarchical modeling.

Behavioral inheritance is supported only by *Javabeans*, EJB and Kevoree where components are described by Java classes following coding guidelines of the models. The similar applies for structural inheritance, which is also supported thanks to Java inheritance. SOFA and Fractal are approaches having their own ADLs, these enable users to reuse structural definition. Structural inheritance is also partially supported with interface inheritance in COM, CCM, OpenCOM, OpenCORBA and DynamicTAO.

Introspection aspect of reflection is well supported. However this is not the case for intercession which is rarely supported and the actual intercession capabilities are usually limited to run-time substitution of components in an architecture, as for example SOFA DCUP does. Fractal and FraSCAti models provide extensible reflection capabilities thanks to the ability to put new controllers into component membrane. Reification of component parts (ports or architecture) is achieved in MetaORB where base-level components are realized as multiple components in meta-level.

Table 2.2 gives an overview of the approaches classified under the frameworks family.

Generative family All surveyed models support specification of component interfaces.They differ in the terminology and the kinds of information they specify. For example, an interface point in

WRIGHT is a port, and in UniCon a player. On the other hand, in C2 the entire interface is provided through a single port; individual interface elements are messages.

Members of the generative family typically distinguish between interface points that refer to provided and required functionality. For example, provides and requires interface constituents in Rapide refer to functions and specify synchronous communication. Finally, WRIGHT and UniCon allow specification of expected component behavior or constraints on component usage relevant to each point of interaction. For example, UniCon allows specification of the number of associations in which a player can be involved.

The static nature of ADLs and UML does not match with reflection and behavioral inheritance well, hence these are not supported.

Structural definition reuse is usually supported in a limited notion of sub-typing or relying on the mechanisms provided by the underlying programming language. For example, ACME supports strictly structural sub-typing with its extends feature, while Rapide evolves components via OO inheritance. AESOP and C2 provide more extensive component sub-typing support. AESOP enforces behavior-preserving sub-typing to create sub-styles of a given architectural style. An AESOP subclass must provide strict sub-typing behavior for operations that succeed, but may also introduce additional sources of failure with respect to its super-class. C2, on the other hand, supports multiple sub-typing relationships among components: name interface,behavior, and implementation. Different combinations of these relationships are specified using the keywords *and* and *not*.

Rapide provides features for refining components across levels of abstraction. This mechanism may be used to evolve components by explicating any deferred design decisions, which is somewhat similar to extending inherited behavior [Medvidovic et Taylor, 2000]. Sub-typing is simply a form of refinement in a general case. This is, however, not true of Rapide of which place additional constraints on refinement maps in order to prove or demonstrate certain properties of architectures. Refinement of components and connectors in Rapide is a by product of the refinement of configurations.

UniCon defines component types by enumeration, allowing no sub-typing, and thus structural inheritance is not supported.

Table 2.3 gives an overview of the approaches classified under the generative family.

Component-oriented languages family All the studied COLs specify external contract of a component with ports which are in the case of ACOEL called *outputs* (required ports) and *inputs* (provided ports). Also, all the languages make it possible to explicitly describe component compositions.

On one hand Java-inheritance in ArchJava, SMALLTALK-inheritance in CLIC, mixin-like decorators in ACOEL and mixinner interfaces in Bichon make it possible to reuse behavior definition. On the other hand, structural inheritance is either not present at all (ComponentJ and Bichon) or it is supported only partially (CLIC, ArchJava, ACOEL). For example, while it is possible to reuse external contract definition in ACOEL or ArchJava, it is not possible to specialize inherited architecture[15].

The goal of CLIC language is to be fully integrated with SMALLTALK and thus objects are considered to be “dirty” components. This, in turn, makes it possible to use SMALLTALK reflection in CLIC.

[15]Architecture specialization and other inheritance operations are detailed in Chapter 4

The similar applies for ArchJava which benefits from Java reflection. A notion of intercession capabilities is present in ComponentJ which supports dynamic reconfigurations of component's architectures.

Table 2.4 gives an overview of the approaches classified under the component-oriented languages family.

	SOFA 2	Fractal	COM	CCM	*Javabeans*
External contract					
required ports	yes - called interfaces	yes - called interfaces	no	yes - called receptacles	no
provided ports	yes - called interfaces	yes - called interfaces	yes	yes - called facets	yes
Architecture description					
explicit composition description	yes	yes	no	no - flat model	no
Inheritance					
behavioral reuse	no	no	no	no	yes - Java inheritance
structural reuse	yes	yes	no - only interfaces inheritance	no - only interfaces inheritance	yes - Java inheritance
Reflection					
introspection	yes - control interfaces	yes	interface discovery, IUnknown	yes	yes - BeanInfo
intercession	partially supported with DCUP	depends on controllers	no	no	no
reification	no	no	no	no	no

	EJB	OpenCOM	OpenCORBA	OSGi	MetaORB
External contract					
required ports	no - only info in Deployment descriptor	yes	yes - called receptacles	dependencies in manifest file	yes - called interfaces
provided ports	yes	yes	yes - called facets	yes	yes - called interfaces
Architecture description					
explicit composition description	no	no	no - flat model	no	yes
Inheritance					
behavioral reuse	yes - Java inheritance	no	no	no	no
structural reuse	yes - Java inheritance	no - only interfaces inheritance	no - only interfaces inheritance	no	no
Reflection					
introspection	yes - EJBMetaData	yes	yes	no	yes
intercession	no	partially - IMetaInterception	yes	no	yes
reification	no	no	no	no	yes

	DynamicTAO	FraSCAti	Kevoree		
External contract					
required ports	yes	yes - called interfaces	yes		
provided ports	yes	yes - called interfaces	yes		
Architecture description					
explicit composition description	yes	yes	no		
Inheritance					
behavioral reuse	no	no	yes - Java inheritance		
structural reuse	no - only interfaces inheritance	yes	yes - Java inheritance		
Reflection					
introspection	yes	yes	yes		
intercession	yes	yes - Controllers	yes		
reification	no	no	no		

Table 2.2 : Frameworks family

	ACME	AESOP	C2	Darwin
External contract				
required ports	yes	yes - called input ports	yes - called interfaces, exported via ports	yes - called services
provided ports	yes	yes - called output ports	yes - called interfaces, exported via ports	yes - called services
Architecture description				
explicit composition description	yes	yes	yes	yes
Inheritance				
behavioral reuse	x	x	x	x
structural reuse	yes	yes	yes	no - just interface inheritance
Reflection				
introspection	x	x	x	x
intercession	x	x	x	x
reification	x	x	x	x

	Rapide	UniCon	WRIGHT	UML
External contract				
required ports	yes - called constituents	yes - called players	yes	yes
provided ports	yes - called constituents	yes - called players	yes	yes
Architecture description				
explicit composition description	yes	yes	yes	yes
Inheritance				
behavioral reuse	x	x	x	x
structural reuse	no - just interface inheritance	no	yes	yes
Reflection				
introspection	x	x	x	x
intercession	x	x	x	x
reification	x	x	x	x

Table 2.3 : Generative family

	ArchJava	ACOEL	ComponentJ	CLIC	Bichon
External contract					
required ports	yes	yes - called outputs	yes	yes	yes
provided ports	yes	yes - called inputs	yes	yes - only one port	yes
Architecture description					
explicit composition description	yes	yes	yes - called configuration	yes	yes
Inheritance					
behavioral reuse	yes	yes	no	yes	no
structural reuse	partially - no architecture specialization	partially - no architecture specialization	no	partially	no
Reflection					
introspection	yes	no	no	yes	no
intercession	no	no	dynamic reconfiguration	yes	no
reification	objects	no	no	objects	no

Table 2.4 : COLs family

2.4 Conclusion

The study made in this chapter shows that approaches of each family interpret components and related concepts differently. Despite this, it is possible to observe that there are certain concepts and mechanisms common to all of them. The concepts are: *components* as unit of encapsulation, *ports* as connection and communication points, *connections* as binding units, *architectures* as composition descriptions and *services* as units of behavior. The mechanisms are: *instantiation* for creating new components according to a description, *service invocation* for communication, *composition* for hierarchical design and *substitution* for run-time adaptation of architectures. We have also observed that there is an attempt to provide a reuse mechanism like inheritance and to provide reflection capabilities like introspection among the studied component-based approaches.

The approaches in generative and frameworks families separate design and implementation. The generative family uses high-level abstractions to describe component-based software and then generates code skeletons into standard OOP languages. The frameworks family also uses OOP languages and forces developers to follow and respect conventions and programming guidelines in order to create component-based software. Separating design and implementation stages and using the *object* concept for building components cause problems in the analysis, implementation, understanding, and evolution of software systems, because conformance between architecture design and final code is not guaranteed [Fabresse *et al.*, 2012 ; Aldrich *et al.*, 2002]. Moreover, mixing the concepts makes developers live harder, because it might be complicated to choose between objects and components to implement a new entity in an application.

The third family, *i.e.* component-oriented languages, bridge this gap between design and implementation by providing conceptual continuum for developing component-based software. However these languages differ in semantics. Besides, the focus they put on different concepts and mechanisms of the component-based approach. Moreover these languages do not pay enough attention to reflection and inheritance which are essential mechanisms for reuse, evolution and maintenance of software.

It is in this context that we propose to present a new language in which we deeply studied the core concepts and the mechanism of component-based approach and to define and build a reflective component-oriented language on top of these.

CHAPTER 3

COMPO's basics

If a system is to serve the creative spirit, it must be entirely comprehensible to a single individual.

Dan INGALLS

Preamble

This chapter introduces the heart of this thesis, a component-oriented programming and modeling language named COMPO. *In the beginning we acquaint readers with the philosophy of the language that have guided us when designing* COMPO. *We try to identify the main concepts and mechanisms of the component-based approach and then in each section we discuss how we think* COMPO *meets these principals. All through this chapter we highlight and argue for the design choices that we have made when building* COMPO.

3.1 The language philosophy

BEYOND rhetoric and the existence of different models and languages, the component-based approach has not yet reach the same level of maturity as, for example, the object-oriented approach has. Therefore our first task was to study existing component-based approaches and make observations. From the study that we have made in the previous chapter, we do have two basic observations:

- The first observation is that there is a gap between the design stage and the implementation stage of component-based development. The gap exists because while during the design stage, usually, the concepts of the component-based approach are available and used. The same concepts are not available during the implementation stage. The traditional programming languages (procedural or object based) are not yet perfectly suitable for component-based development in every-day practice. They require programmers to respect conventions or design patterns to implement the concepts and mechanisms on which the component-based approach stands. For example, when programming a *Javabeans* component, one has to follow the Java language naming conventions and use the Observer design pattern [Gamma *et al.*, 1995a] (*see* Section 2). This complicates the implementation but also the testing, maintenance and evolution of the source code of the application. The main reason for this is that the used programming language does not allow to simply express and use the basic concepts and mechanisms of the component-based approach.

- The second observation concerns the existing component-based programming languages such as ArchJava, ComponentJ, Lagoona or Piccola. Although these languages effectively integrate some concepts and mechanisms related to component-based approach, they do not all offer the same, even if they use a common vocabulary. For example, both ArchJava and ComponentJ present mechanisms that allow the building of composite components, however, in ArchJava it is not possible, for instance, to export the behavior of internal components and to define new component structures at runtime. Unlike ComponentJ, whose components are used to instantiate objects, ArchJava's components hold state variables, implemented methods and communication ports. By doing so, dynamic construction of component structures is only allowed within pre-established connection patterns. We believe that the significant reason why is it so is a missing definition of the core component concepts and mechanisms [Fabresse *et al.*, 2008].

The first observation hints that there is a need for a component-oriented language (COL) that allows component-based developers to express themselves easily, in all stages of development, without the necessity to switch into lower-level concepts. As emphasized by J. Privat in his thesis [Privat, 2006] "*a good programming language should allow the programmer to express themselves easily, so it should be as close as possible to the human way of thinking.*" The focus on higher-level concepts improves expressiveness of the language and lets the language users to easily preserve original architecture designs in all stages of development process. In other words, it is possible to bridge the gap between design and implementation stage by using higher-level concepts for all stages of development Also, the presence and usage of higher-level concepts make solutions more understandable and therefore the later evolution and maintenance easier.

The second observation suggest that the disparity between existing COLs calls to better identify and define the core concepts and mechanisms of the component-based approach. The results of our predecessor[Fabresse, 2007 ; Fabresse *et al.*, 2012] and the synthesis of existing component-based approaches made in Chapters 1 and 2 shows that there are some concepts and mechanisms that are common for relevant majority of component-based approaches. In order to facilitate a better comprehension and in accordance with the objectives of this thesis, we define **the core concepts and mechanisms** of the component-based approach as follows:

- Concepts:

  Component - a run-time entity which provides and requires services through ports.

  Descriptor - an entity which describes the structure and the behavior of a particular kind of components in terms of declaration and definition of the external contract and the internal architecture.

  Port - a named communication and connection point; described by a name and a list of service signatures.

  Service - a unit of behavior definition.

  Connection - describes a binding from one to another port.

- Mechanisms:

  Component creation (instantiation) - a mechanism for building new components according to the description a descriptor defines. Such components are then called instances of the descriptor.

  Service invocation - a mechanism for run-time communication in between components

  Composition mechanism - a mechanism for creating a new component by connecting off-the-shelf components within the context of the new component

  Substitution mechanism - a mechanism for replacing components

Having the definition of the core concepts and mechanisms, we define a component-oriented language:

Definition 1 (Component-oriented language (COL)) *A language used to design and implement software components (*development for reuse*) with well defined external contracts; that can be stored in libraries (also called shelves) and, in the same time, to develop applications by assembling off-the-shelve components, that is, to allow to describe software architectures in terms of connecting components selected from libraries (*development by reuse.*)*

The above paragraphs describe our intentions and desires we have in minds while designing COMPO. In the rest we define all the core concepts and mechanisms we have described. For each specification of a concept or a mechanism, we have adopted a constructive approach, that is, we construct the specifications driven by the knowledge we have gained in Chapter 2. We believe that this approach helps to fulfill the, so called, "COMPO's philosophy", which can be summarized into the following:

COMPO's philosophy

Keep the language as simple, minimal and uniform as possible, while in the same time incorporate all core concepts and mechanisms necessary for description and implementation of independent components and for description and implementation of high-level component-based architectures.

3.2 Concepts

This section present the core component-based concepts. For each concept we present the general motivation, definitions and design choices made for COMPO.

3.2.1 Components and Descriptors

Historically, the nowadays object-oriented languages went through the never-ending debate, whenever the languages should be class- or prototype- based [Lieberman, 1986a ; Abadi et Cardelli, 1996]. It seems that the victory was claimed by the class-based languages, which are now very widespread, although prototype-based languages dominate in specific domains, such as web and Javascript [Flanagan, 1998]. In the world of objects, the terms *class* (or descriptor) and *instance* denote the object descriptions in the code and the objects themselves at run-time.

By analogy, one may ask whether a COL should be built on the descriptor basis [1] or on the prototype one.

Chapter 2 has shown that in opposite to the object world the component world has a problem with terms for descriptors and instances. The ambiguity reflects in vocabulary differences in literature and languages. For example, a descriptor is referred to as "component class" (keyword `component class`) in ArchJava while the same concept is named "component" (keyword `component`) in ComponentJ. Thus, when we speak about a "component", we speak about an instance of a component class in ArchJava and about a descriptor in ComponentJ.

Apart from this terminology problem, Chapter 2 has also shown that the majority of component-based models is descriptor-based, not prototype-based. In fact, we are aware of only one prototype-based COL proposed by [Zenger, 2002]. Component's in that work have neither state nor identity and they are created through *refinement* primitives of existing components. The bootstrap component is called `component` and it provides and requires no service. The services of a component can not be used directly and the latter must first be *instantiated*. Indeed, the language distinguishes the notions of *component* and *component* instance. However, this proposal has the merit of raising the question of whether a COL can be (or should be) prototype-based since most languages (ArchJava, ComponentJ, etc..) are descriptor-based.

The arguments for or against the use of descriptors in the world of components seem similar to those in the world of objects [Dony *et al.*, 1992].

[1] We prefer to not use the "class" term to avoid ambiguity with the object world.

The arguments in favor of prototype-based languages are:

- simplicity: this argument was not accepted in the world of objects as most object-oriented languages are descriptor-based and does not seem to prevail in the world of components;
- independence - prototypes are not associated with a descriptor: this argument does not constitute a limitation of descriptor-based approach in the component world since a component is packaged as an archive containing a particular descriptor (*cf.* Chapter 2) to be easily deployed or put on shelf.

In contrast, drawback of the prototype-based approach is the difficulty to find the right place where to store all what is common to members of a given family and should not be duplicated, for example a link to a given method. The global dynamic handling of what is store by a given family is still a challenging issue. Although satisfactory solutions have been proposed, such as *traits* in SELF [Ungar et Smith, 1987]. In order to make large applications, this lack of abstraction and thus structuring is a strong limitation, therefore in COMPO, we chose the descriptor-based approach.

Choice 1 *A* component *is a run-time entity, instance of a* descriptor*, which provides and requires services via ports.*

The choice to design COMPO as a descriptor-based language raises three questions.

The first question: *Who do we put on the shelf and reuse later, the descriptors or components?* First, we recall that as a shelf we consider here an archive library (jar files, for example) containing reusable software entities in different architectures (applications). In most of the component-based approaches, the archive put on a shelf contains one or more descriptors. Indeed, a component is a run-time entity and it can be hardly put on the shelf for reuse. However, the *Javabeans* model offers the possibility to include a serialized *Javabeans* component (saved to disk) in an archive. This allows to put on the shelf already initialized (with values) components. In other approaches, the initial values can be specified in a configuration file included in the archive and read during the instantiation of the descriptor. Both approaches therefore offer the same opportunities and in COMPO we chose to put on the shelf only descriptors. If we had chosen the approach proposed by the *Javabeans* model we would have had to deal with the archives containing descriptors and those containing components differently.

Choice 2 *The descriptors can be placed on the shelf.*

The second question: *Can we combine descriptors or not?*

An assembly of components is a set of inter-connected components. There are basically two ways how to describe an assembly: either we describe how components will be connected or we combine descriptors [Lau et Wang, 2005b ; Lau et Wang, 2005a]. For example, while *Javabeans* and ArchJava enable to describe an assembly of components only, ComponentJ and Scala [Odersky et Zenger, 2005] can combine descriptors in order to define an assembly. In Scala, descriptors are classes and to create an assembly of components is essentially a mechanism for combining classes based on *mixins*. It is

typical for these approaches that the instantiation of a descriptor is possible only if all requirements are satisfied. These models are generally safe, but prohibit late connections between components, because requirements cannot be satisfied later. In contrast, in ArchJava, all requirements does not have to be satisfied if only a part of its functionality is used in the application.

Moreover, it seems that combining descriptors is more the matter of static design then the matter of run-time. For example, in object-oriented languages supporting multiple inheritance, a class (*i.e.* a description) could be defined as a combination of multiple super-classes. In opposite,to create an assembly of instances (*i.e.* components) is a matter of run-time and it offers better perspectives considering the dynamical aspects of languages.

We believe that an ideal COL should offer a unified combination mechanism for both descriptors and their instances. There is currently no language that integrates such a mechanism but it seems that this goal could be addressed by a reflection approach, where descriptors would be just a special kind of components. In this thesis we design a reflective language and based on the above observations we have made the two following choices:

Choice 3 *A component (not a descriptor) is a subject for connecting.*

Choice 4 *Descriptors cannot be combined.*

The third question: *Should architectures be separated from descriptors?* Before we answer this question, it is essential to present our understanding of the architecture of a component:

Definition 2 (The architecture of a component) *The architecture of a component is a description of an internal composition, i.e. a system of internal components and their inter-connections, according to which the component will be initialized.*

Indeed, two components could have the same architecture and a component could have more architectures. For example, SOFA separates architectures and component-types (a component-type defines the external contract, *i.e.* provided and required ports), thus, two architectures could implement the same component-type, hence the type is reused, but an architecture cannot be reused for two different component-types. For example, a pipeline architecture implementing a filter component-type (with one input and one output port) cannot be used as an implementation of a dispatcher component-type (with one input and two output ports.) In general, it seems that architectures are coupled with "component-types" and their separation does allow for type reuse but not for architecture reuse. Moreover, the separation may cause incompatibilities in case one of the two (an architecture and its component-type) evolves. We think that the reuse of architectures description could be achieved differently, for example with inheritance. With COMPO we follow the majority of component-based approaches and we chose the following:

Choice 5 *The description of the architecture of a component is a part of a descriptor of the component (i.e. it is not separated).*

By answering the three questions we are now ready to define a COMPO-descriptor:

Definition 3 (COMPO-descriptor) *A descriptor defines the structure and behavior of its instances called components. The behavior is defined by a set of services definitions. The structure is defined by description of ports and by description of the architecture of its instances. Descriptions of external (resp. internal) ports define an external contract (resp. an internal contract) of instances of the descriptor.*

The definition was described in MOF [OMG, 2011a][2] and it is visualized in the Figure 3.2.

A descriptor owns several `PortDescription` entities in order to describe the external and internal contract of its instances. It also owns zero or more `ConnectionDescription` entities in order to capture the the architecture of its instances. Finally it owns zero or more `Service` entities to capture the behavior of its instances.

In fact, the external and internal contract together with the architecture specification define an Architecture Description Language, as it is shown in Figure 3.1. In the following we will show that COMPO contains all concept necessary for architecture description. Later in Chapter 5 we will show the descriptors can be used for generating code, as it is done by standard ADLs like WRIGHT or Fractal ADL. Descriptors facilitate the modeling aspect of COMPO language.

Fractal ADL

```
<definition name="HTTPServerType">
  <interface role="server" signature="IRunnable" nam
/>
</definition>

<definition name="HTTPServer" extends="HTTPServerType">
  <component definition="FrontEnd" name="fE" />
  <component definition="BackEnd" name="bE" />

  <binding client="fE.backEnd" ser
  <binding client="this.default" s
</definition>
```

Compo

```
Descriptor HTTPServer {
  provides {
    control : { run(); status(); }
  }
  internally requires
  {
    fE : FrontEnd;
    bE : BackEnd;
  }
  architecture
  {
    connect fE to default@(FrontEnd.new());
    connect bE to default@(BackEnd.new());
    connect backEnd@fE to default@bE;
  }
  …
}
```

Figure 3.1 : A parallel between descriptors and ADLs.

When it is declared that an instance of a descriptor offers a service, but the service is not defined, then the descriptor is considered as an *abstract component descriptor*. Abstract descriptors cannot be instantiated and they are present for code factorization and reuse purposes.

[2] MOF is a graphic language, the core of UML, used for definition of Domain Specific Languages

Ports and connections between them are created according to the `PortDescriptions` and `ConnectionDescriptions`, *i.e.* they realize[3] them. More about ports and connections between them can be found in Sections 3.2.2 and 3.2.4

[3] In UML modeling, a realization relationship is a relationship between two model elements, in which one model element realizes the behavior that the other model element specifies

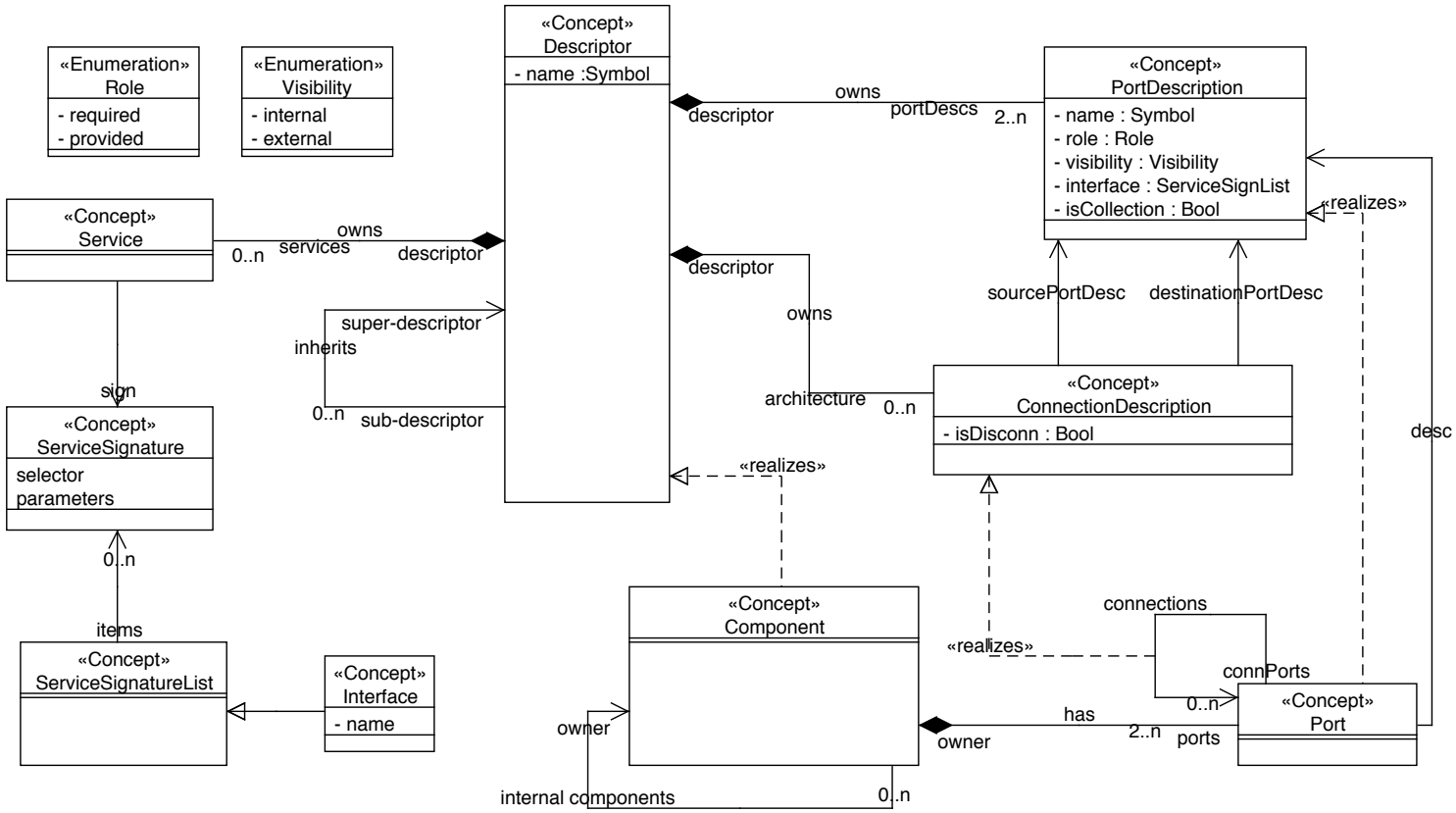


Figure 3.2 : Descriptor concept definition in MOF language

Definitions 2 and 3 together with the Choice 1 imply that a component might be composed of other components called *internal components*. Such a component is than called a *composite* and we will discuss them in detail in Section 3.3.3.

The HTTPServer example

At a glance, the Listing 3.1 shows a definition of a descriptor named `HTTPServer` modeling very simple HTTP servers. It defines a `default` provided port through which it provides the services `run` and `status`. It states that a server is composed of two internal components, an instance of `FrontEnd` accessible via the internal required port `fE`, and an instance of `BackEnd` accessible via the internal required port `bE`. These internal components are connected together so that the front-end can invoke services of the back-end. The `HTTPServer` descriptor explicitly defines the implementation of the `status` service. The provided service `run` is implemented by a delegation connection to the provided port `default` of the front-end. Figure 3.3 shows a diagram that represents a component, instance of the `HTTPServer` descriptor.

```
Descriptor HTTPServer {
  provides {
    default : { run(); status() }
  }
  internally requires {
    fE : FrontEnd;
    bE : BackEnd;
  }
  architecture {
    connect fE to default@(FrontEnd.new());
    connect bE to default@(BackEnd.new());
    delegate default@self to default@fE;
    connect backEnd@fE to default@bE;
  }
  service status() {
    if(fE.isListening())
    {
      return name.printString()  + ' is running'
    }
    else
    {
      return name.printString() + ' is stopped'
    }
  }
}
```

LISTING 3.1 : The `HTTPServer` descriptor.

Let's look at each point more precisely. A descriptor defines the *structure* and *behavior* of its instances. The behavior is given a set of services definitions, for example a part of an `HTTPServer`'s behavior is defined with the `status` service. The structure is given by descriptions of ports and connections. Descriptions of external (resp. internal) ports define an *external contract* (resp. an *internal*

contract). For example the external contract of `HTTPServer` instances is defined by the declaration of the provided port `default` and its internal contract is defined by the declaration of the `fE` and `bE` internal required ports .

A component may be composed of (*internal*) components (e.g. a HTTPServer is composed of an instance of `FrontEnd` connected to an instance of `BackEnd`) and it is then called a composite. A composite is connected to its internal components via its internal required ports. The services of a composite can then invoke the services of its internal components through such ports. The system composed of internal components and their connections is called the *internal architecture* of a composite. An example is given in the `architecture` section in Listing 3.1.

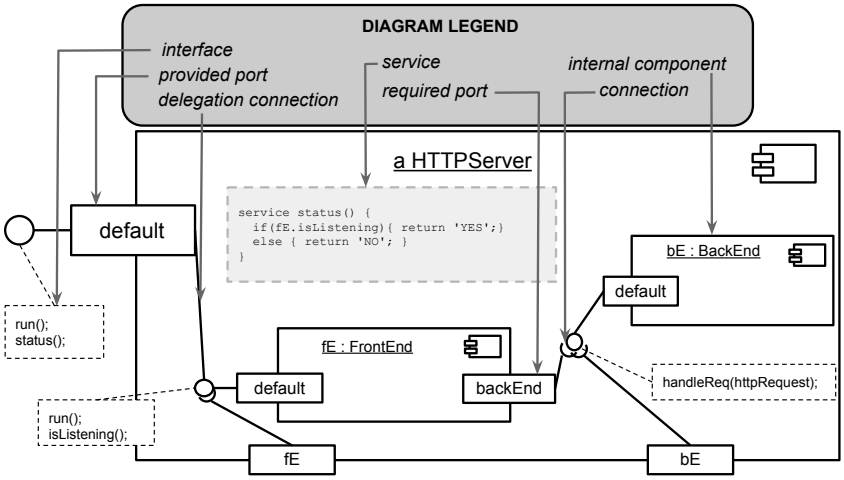


Figure 3.3 : The diagram shows a logical representation of an instance of the HTTPSERVER descriptor presented in Listing 3.1, after it has been created and initialized.

3.2.2 Ports

In the beginning of this chapter we have defined ports as named communication and connection points described by lists of service signatures. In this section, we will try to determine the nature of these communication and connection points and their exact role in the world of components.

The original idea behind ports was to strengthen the encapsulation of components. A component is seen as a capsule which cannot be acceded otherwise than by one of its ports (or interfaces).

> "*a component can only be accessed through well defined interfaces*" [Szyperski, 2002]

This explicit description of component's contract increase the independence of components from their environment (other components) because limits the number of connections and also restricts the communication by stating which services are provided and required.

The terms *port* and *component interface* are often confused with the term *interface* in the literature. In the following, we understand the terms *port* and *component interface* as labels for the concept

of connection and communication point. While the *interface* term represents the concept of the contract description (syntactic, behavioral, quality) of a port, often used for assembly verification or for validating the use of a component.

The study of existing approaches has shown that ports might be classified as *unidirectional* and *bidirectional*. Through unidirectional ports a component either provide services or require services, such a port is than called *provided* or *required* port. ComponentJ or Fractal are representing the unidirectional understanding of ports. By contrast ArchJava or UML represent the bidirectional approach where services might be both provided and required through a port.

In both cases, a port defines a view-point and a security policy for a component. Required ports define the views that the component may have on external components while provided ports define views that external components can have on this component. Similarly, when a component is accessed via one of its ports, the port is the guarantor of a security policy, *i.e.* the client components may only use the services available through this port. By allowing to mix required and provided services within the same port, the bidirectional approach can describe more accurately the dependencies and collaboration (in the sense of UML) between components.

Although this may seem restrictive at first, we chose to integrate unidirectional ports for COMPO. This choice is essentially motivated by the non-anticipation politics of our predecessor SCL. Indeed, using two ports, one required and one provided, instead of a single bidirectional port in the design of a component seems to be redundant, but it lets the architect to decide whether the component that will use the provided services provided is the same that the one that will provide the required services. By declaring a bidirectional port, it is imposed that both required and provided services will be provided and required by one client. That could be considered as a unnecessary restriction for usage contexts. Our idea is to provide simple and as little as possible constrained components, so that they can be adapted and reused in different contexts. The unidirectional ports are also easier to understand and use in practice. In addition, if necessary, it is still possible to emulate bidirectional ports with two unidirectional ports.

Choice 6 *A component has* unidirectional *ports.*

Following this choice, the definition 4 sets up the vocabulary we use.

Definition 4 (Required (resp. provided) port) *Required (resp. provided) port of a component is a named connection and communication point through which the component requires (resp. provides) a set of services.*

The definition raises a new question: *How should we describe the concrete contract of a component?* In general, interfaces are used to specify how components can be assembled or used in an architecture. interfaces are defined either at a local level (associated with a port) at a global level (associated with a component). According to [Beugnard *et al.*, 1999] interfaces for contracts definition are generally classified into four levels

Syntax : These contracts specify the signatures of the provided and required services (name, parameters, result, exceptions).

Behavioral : These contracts specify how a set of services can be used .

Synchronization : These contracts are required in a distributed and/or concurrent context to specify the behavior of components in terms of synchronization between service invocations.

Quality of service (QoS) : These contracts are usually crucial in the field of embedded or real-time systems to ensure quality constraints in an overall architecture.

Component-based approaches offer formal or informal notations tailored to their needs to describe contracts at these levels. Syntactic contracts are simpler and can be defined using an interface definition language, such as *Interface Definition Language* (IDL), or directly with the notion of *interface* in the object-oriented languages, *e.g.* interfaces in Java. Moreover, the mature component-based languages like Fractal, ArchJava or ComponentJ are generally limited to the syntactic contract level. In ArchJava, an interface specifies for each service if it is provided or required, because each port is bidirectional and is described by a single interface. SOFA supports behavioral contracts with a formalism based on regular expressions to be able to describe the sequences of invocations of valid services (*protocols*.) Such a formalism has also been used to describe web services protocols [Tremblay et Chae, 2005]. For example, a component dedicated for networking provide the following three services: `open(adr)` (opens a network connection for the address specified by the `addr`), `send(data)` (sends data `data` through the connection) and `close` (close the connection). A protocol for such a component used to describe the order of invocation of these three services for a valid use is: `open` (invoked only once), then `send` (as many times as necessary) then `close` (once) which can be expressed as `open;send*;close`. Other formalisms can be used to express behavioral contracts as state machines of UML, automata languages [de Alfaro et Henzinger, 2001] or symbolic protocols [Pavel *et al.*, 2005]. The synchronization and QoS contracts are poorly supported in the existing component-oriented languages and component frameworks. These kinds of contracts are addressed in ADLs, which are dedicated to the specification and not to directly produce an executable application.

From the syntactic point of view, we see that the compliance between interfaces is often determined by the compatibility of the types that have been associated with them. When two components are connected via their ports, it implies that the types of their interfaces are compatible. For example, a required port typed by a I_1 interface and connected to a provided port typed by a I_2 interface supposes that the type defined by I_1 is a super-type of the type defined by I_2. There are generally two kinds of type systems: those based on names (*named type systems*), *e.g.* Java, and those based on the structure (*structure type systems*), *e.g.* Objective CAML [E. Chailloux et B. Pagano, 2004]. The use of names makes it easier to capture the semantics [Büchi et Weck, 1998]

> " *[...] types stand for semantical specification. While the conformance of an implementation to a behavioral specification cannot be easily checked by current compilers, type conformance is checkable. By simply comparing names, compilers can check that several parties refer to the same standard specification.* "

The structure type systems [Cardelli, 1997] are less expressive but offer better decoupling between entities because the sub-typing relation is derived from the structure of interfaces and not from a common name. For example, to specify that "a component requires a stack" is more semantic than

"a component requires two services `push` and `pop`." However, in the first case, there must be a stack interface (global) defined and the component may not be connected to any component but only to components providing the same stack interface or one of its sub-types. In the second case, an interface may be a sub-type of another even if there is not a direct relationship, because the sub-typing relation is derived from the structure.

In COMPO, we chose to integrate a simple model regarding the descriptors. Thus, an interface is local and attached to a port.

Choice 7 *A* interface *is associated with a port.*

We restrict ourselves to check the syntactic level of contracts.

Choice 8 *A* interface *specifies a set of signatures of services. The* interface *compatibility is based on sub-typing relationship between their types which is based on the inclusion of sets of signature services.*

An interface does not need a name in COMPO and sub-typing relationship is structural. This decision is motivated by our desire to decouple components. Indeed, with a type system based on names, two components can only be connected if the types of their interfaces are directly related. A structural approach seems to offer greater independence components in their definition at the expense of semantics as we have seen. This view is sometimes called the "*Duck typing*" [Anantharam, 2001]

> " *This method . . . [of] . . . just relying on what methods it supports is known as "Duck Typing", as in "if it walks like a duck and quacks like a duck . . . ". The benefit of this is that it doesn't unnecessarily restrict the types of variables that are supported. If someone comes up with a new kind of list class, as long as it implements the join method with the same semantics as other lists, everything will work as planned.*"

However, for the reuse purposes, we support the global named interfaces in COMPO. The users can define named interfaces by use of `interface` statement. These interfaces can be associated with ports. The interface compatibility rules apply also for named interfaces, *i.e.* their conformance is derived from their structure.

In COMPO, ports realize port descriptions (similarly to slots realizing classes' attributes in UML [OMG, 2011c]). A port description define the *name*, *interface*, *role* (provided or required) and *visibility* (external or internal) of a port. A port description also define whenever a port is a collection port or not. In the following we describe each of these aspects in detail.

Names The name of a port is a standard identifier conforming to the following regular expression `[a-z][a-zA-Z_0-9]*`. In COMPO, the ports names are unique identifications used to refer to a particular port. The uniqueness means that in the definition of a descriptor it is not possible to define two ports with the same name.

Interfaces The interface of a port is a set of service signatures which could be given in three forms:

- as an explicit list (we call such a list an *anonymous interface*), for example the `default` port declaration in Listing 3.1
- as a named interface, e.g. `printer : IPrinting` where the interface `IPrinting` was created with the statement: `interface IPrinting {print(text); ....};`
- as a descriptor name (e.g. `cd`); in this case, the list is the list of signatures of services associated to `cd`'s default provided port (the `fE` port declaration in Listing 3.1 is an example).

A special case of a named interface is the universal interface $*$ which we introduce by the following definition:

Definition 5 (Universal interface $*$) *In case of provided ports, the universal interface $*$ means that a port offers all services already provided by the descriptor of a component that owns the port. In case of required ports, it means that any service could be invoked through such a port.*

Visibility The visibility of a port specifies whenever the port can or cannot be accessed from the outside environment of the component which owns the port. There are two basic visibilities of ports: *external* and *internal*. External ports are visible from the outside environment and are used for communicating with neighboring components in the environment. Internal ports of a composite are used for communication with internal components, *see* Section 3.3.3 for more details. Internal ports and the internal architecture of the owner are not accessible from the outside environment. If the visibility of a port is not specified, then the port is by default external.

Roles Since we choose to use unidirectional ports in COMPO, we have to impose port roles to be able to distinguish ports directions. The role *provided* (resp. *required*) specifies the direction of a port, *i.e.* it represents the fact that the port offers services to (resp. demands services from) the environment. Unfortunately, there is no consensus in the literature on which terms should be used for roles. For example Fractal use "server" (aka provided) and "client" (aka required) role names while SADL "input" (aka provided) and "output" (aka required) role names. Required ports are communication points through which a component invokes services it requires. `fE.isListening()` is an example of a service invocation expression in the code of the `status()` service defined in the `HTTPServer` (*cf.* Listing 3.1) descriptor, made through the `fE` required port. These required services are provided by components connected to required ports. A component provide services through provided ports making them accessible from outside.

Collection port Collection ports address the problem of multiple relationships between components. For example, the relationship between a `Bank` component and `Client` components, where a bank may have many clients. Support for multiple relationships between components make it possible to design dynamic architectures where the number of connections between component vary during application lifetime. For example, the `BackEnd` of the `HTTPServer` component shown in Figure 3.3 may need to dynamically create new requests-handler components, one for each HTTP request.

Two approaches described in Section 2 address the question of the multiple relationships:

- SOFA (*see* Section 2.2.2) and Fractal (*see* Section 2.2.2) allow to set a multiple cardinality for an interface of a component. The cardinality is constant value during computation. When the code for such an interface is generated, an attribute of type table or list is created.
- ArchJava (*see* Section 2.2.4) allows for dynamic addition of ports through the notions of *port interface* and *connect pattern*. Thus, for each new component `Client` , the component `Bank` is automatically equipped with a port for the link to the new component.

The solution we propose for COMPO represents a compromise between these two.

Choice 9 *A component can have* collection ports.

Definition 6 (Collection port) *A named and ordered collection of required or provided ports. Each port of the collection can be accessed by an index.*

By this choice 9 and the definition 6 we state that a COMPO component can own *collection ports*. Collections ports are declared by putting empty square brackets after the name of a port. For example, the statement `marshalling[] : {serialize(); materialize();}` defines a collection port named marshalling through which services `serialize` and `materialize` are accessible. However, the size of these collections (*cardinality*) is not set by the programmer unlike SOFA and an COMPO interpreter may decide to adopt the on demand allocation policy, such as ArchJava. In the following we use the term *collection port* to denote a collection of ports. This is a misnomer because a collection of ports is not a port. However, this term is intuitive and it makes reading easier.

In general, collection ports resemble arrays in standard programming languages. In theses languages the developers often use other kinds of collections, for example unordered collections like stacks and dictionaries. For these collections we do not have a component-based analogy. Nevertheless, our reflection level (presented in the Chapter 5) enables COMPO's users to define new kinds of ports and so, a new version of collection ports could be designed to simulate unordered behavior. The question of unordered collection ports remains open as a perspective for the future development of COMPO.

In OOP, it is very common to have a *sizeof* operator to determine a size of an object, array, type, etc, for example `sizeof` in C# or PHP. In accordance, we provide **`sizeof`** operator to determine the count of connection a collection port has. For example, suppose that `items[]` is a collection port with 5 connections to 5 item components, then the expression `sizeof(items)` returns 5.

Definition 7 (The `sizeof` operator) *The* **`sizeof`** *operator, when applied on a port, returns the count of connections the port participate in.*

Default port By default, every component owns one externally provided port named *default port* through which it offers its all services which have already been provided. This solution directly comes from SCL and it unifies the vision of objects and components in our model [Fabresse, 2007]. Objects are seen as primitive components equipped with a single external provided port through which they provide their methods (called services.)

Choice 10 *Every component has a port named* `default` *through which all services already provided by the component are available. The instantiation mechanism of descriptors returns a reference to the* `default` *port of the newly created component.*

Since all components have the `default` port we usually omit it in our figures.

Graphic conventions Figure 3.4 presents the UML-like graphical conventions used in the rest.

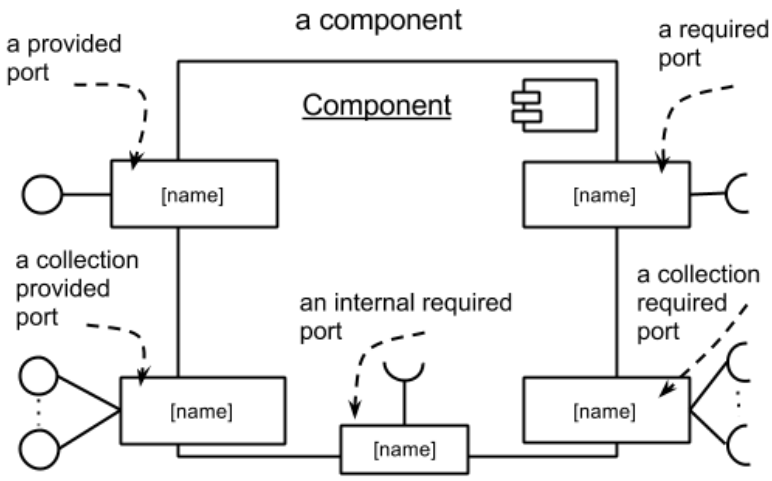


Figure 3.4 : Overview of the UML-like graphic conventions used for COMPO

Example in COMPO

In the HTTPServer example (*see* Section 3.2.1) we have presented a HTTP server component which receives HTTP requests from network, processes the requests and finally it create and send the responds. While the `FrontEnd` component of the server is responsible for receiving requests, the (BackEnd) component process and creates responds for the requests. For each request, it creates a `RequestHandler` component. The BACKEND descriptor represent an example of a dynamic architecture, because for each HTTP request an instance of the REQUESTHANDLER descriptor is created.

Listing 3.2 shows the BACKEND descriptor. The descriptor specifies one provided port named `reqHa`[4], providing the `handleReq(r)` service. Internally, it defines three internal required ports. The

[4]The "reqHa" abbreviation stands for request handler.

```
Descriptor BackEnd {
  provides {
    reqHa : { handleReq(httpRequest); }
  }
  internally requires {
    analyzer : RequestAnalyzer;
    logger : Logger;
    handlers[] : RequestHandler
  }
  architecture {
    delegate reqHa to inReqHa@analyzer;
    connect logger to default@(Logger.new());
    connect analyzer to default@(RequestAnalyzer.new());
    connect logger@analyzer to logging@logger;
  }

  service addHandler()  {
    |i|
    i := connect handlers to default@(RequestHandler.new());
    connect outReqHa@analyzer to reqHa@handlers[i];
  }
}
```

LISTING 3.2 : The BackEnd descriptor

```
Descriptor RequestAnalyzer {
  provides {
   inReqHa : { handleReq(req, index); }
  }
  requires {
    logger : { log(str) };
   outReqHa[] : { handleReq(httpRequest); };
  }
  ...
}
```

LISTING 3.3 : The RequestAnalyzer descriptor

first port is named `analyzer` and described by the interface of the default provided port (*see* Section 10) of the REQUESTANALYZER. The second port is named `logger` and described by the interface of the default provided port of the LOGGER. The third port is a collection port named `handler` and described by the interface of the default provided port of the REQUESTHANDLER. In the architecture section, we declare one delegation connection and three regular connections (*see* Section 3.2.4 for more details about delegation and regular connections.) The delegation connection says that the external provided port `reqHa` delegate service invocations to the `inReqHa` port of the internal component connected to the `analyzer` internal required port. The first two regular connections connect new instances of LOGGER and REQUESTANALYZER descriptors to the internal required ports `logger` and analyzer respectively. The last regular connection defines that the two internal components are interconnected between the `logger` and `logging` ports. Finally we can see the implementation of the service `addHandler` service which dynamically adds and connects new instances of the REQUESTHANDLER descriptor. The newly created components are connected to the `handlers` internal required collection port and then each newly created component is connected to the `outReqHa` external required collection port of the `analyzer` component, which is an instance of the REQUESTANALYZER shown in Listing 3.3.

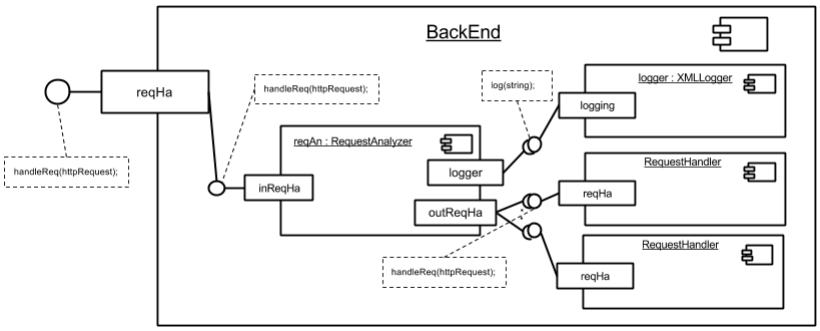


Figure 3.5 : An example of a dynamic architecture with a collection port in an instance of `BackEnd`

3.2.3 Services

The services are units of behavior (*functionalities*) which a component may provide, for example a simple math component may provide services for addition, subtraction, multiplication and division. In some systems, a component has only one functionality. For example, a *Unix process* can be considered as a component performing operations on input (*stdin*), output (*stdout*) and error (*stderr*) streams. Two processes can be combined by redirecting the output stream of the first to the standard input stream of the second. Such an architectural style is often called *pipes and filters* [Shaw et Garland, 1996].

However, in the majority of component-based models, a component has several functionalities that are represented by *services*. A service is generally a function (or an operation) defined by a component that has a name, parameters[5] and a result. In some models, a service is defined as a set of

[5]The term "parameter" (or "formal parameter") can refer to a variable bound in a lexical closure, *e.g.* x is a parameter in

functions. For example, a service account management can be divided into three "sub-services": consultation, withdrawal and deposit.

The *service invocation* is the term used in this thesis to describe the mechanism (presented in a more comprehensive and detailed way in Section 3.3.2) of a component to run a service as an effect of receiving an invocation and to emit invocations of services to other connected components. This mechanism should not be confused with the *message sending* mechanism in the the world of objects. As we will see later, the difference with the message sending is related to the specific assembling mechanism in the world of components. The following definition captures the consensus made by the majority of the current models:

Definition 8 (Provided service) *a functionality defined in the source code of a component, which is offered to other components through ports, so they can invoke it later.*

In general, provided services resemble *public methods* from the object-oriented world. A descriptor introduces services to specify functionality of its instances. When a service is listed in the interface description of one of its provided ports, then the service is public. The descriptor may also define services which are not provided through ports in order to factorize its implementation. Such services then resemble *protected methods* from the object-oriented world.

Definition 9 (Internal service) *a service which is not offered trough any provided port of the component (that defines it) is not accessible from outside of the component.*

A descriptor also expresses, via required ports, the services that its instances require from other components. These *required services* are not defined by the descriptor of the component, but they may be invoked in its code, *i.e.* the provided and internal services may invoke required services.

Definition 10 (Required service) *a service which is necessary for implementation of the behavior of a component* `X` *(invoked in the code of its services), provided by an external component connected to the required port of the component* `X`.

In COMPO, services are defined inside descriptors. Each service has a signature given by the following template: `<selector> (<parameter1-name>, <parameter2-name>, ... )`. A definition of a service consist of the **`service`** keyword followed by the *service signature* and a source code written in brackets after the signature, for example, *see* the service `add` in Listing 3.4.

Definition 11 (Service signature) *Each service has a signature given by the following template:* `<selector> (<parameter1-name>, <parameter2-name>, ... )`. *Two service signatures are compatible if they have the same selector and the same count of parameters.*

the following function definition *f (x)* ..., while the term "argument" (or "real parameter") represents the value substituted for a parameter when such a function is used, *e.g. 3* is an argument in the following function call *f(3)*.

At run-time, services have access to ports and architecture of an instance of the descriptor (component) they are associated with and thereby they are able to control the state of the instance.

The syntax of COMPO is mainly a java-like syntax with a few Smalltalk syntax constructs which we kept alive of SCL (the predecessor of COMPO). To describe the desired functionality, the body of services can be composed from the temporal variables definition like `|temp, sum|`; the standard program flow control structures like **`if-else`**, **`for`**, etc.; the service invocation expression like `calc.add(1,1)` or the connection statement **`connect-to`**. The complete grammar can be found in Appendix A We have chosen the java-like syntax, because we consider it more readable and expressive for structural descriptions, and hence more suitable for architecture descriptions.

Example in COMPO

In the following example (*see* Listing 3.4) we present the descriptor CALC of a very simple `calculator` component providing three arithmetic services and one probability service. `calculator` provides these services through ports `arithmetic` and `probability`. To fulfill its commitment to provide the probability service `rand`, the component requires a random generator via required port `randGen`. The definitions of services `add` and `mul` show the use of arithmetic operators and value return. Service `pow` shows program flow control structure **`for`**, assignment and a service invocation. An example of a required service invocation is captured in the definition of the `rand` service. Listing 3.5 and Figure 3.6 show an example of how the `calculator` component can be instantiated and used.

3.2.4 Connections

Connections represent the central concept for binding in the component world (*see* Section 2) and they are present in various forms in the existing component-based proposals. In general, a component has ports in order to be able to establish connections to ports of other components. In COMPO, there is not other way how to bind two components than to establish a connection between their ports. In the following, when we say that two components are connected, then it is an admitted shortcut to say that a port of the former component is connected to a port of the later component.

The literature presents two basic approaches to bind components: n-ary and binary connections. For example, ArchJava offers primitive n-ary **`connect`** statement to bind a set of ports directly, while Fractal offers primitive binary **`bindFc`** that binds a required port to a provided port. Although these approaches include respectively bidirectional and unidirectional ports, they raise the following question: *Should connections be binary or n-ary?*

In COMPO, we chose to provide binary connections, because they are less prone to ambiguity existing when n-ary connection are present, as it is for example in ArchJava where there might be more provided service candidates for one required service. Furthermore, the n-ary connections can be factorized into binary connections in most cases. The only cases when this is not possible are those where it is needed to combine services provided by different provided ports to connect them to a single required port. These cases can be easily treated with adapter components [Gamma *et al.*, 1995a]. Moreover, in the case of binary connections, the validity of a connection between ports is easy to verify in opposite to the n-ary case, since it is based only on the compatibility of two interfaces that are associated with them.

```
Descriptor Calc {
  provides {
    arithmetic : { add(x, y); mul(x, y); pow(base, exp) };
    probability : { rand() };
  }
  requires {
    randGen : { getRandVal(seed); }
  }
  service add(x, y) { return x + y }
  service mul(x, y) { return x * y }
  service pow(base, exp) {
    | res i |
    res := 1;
    /* the exp times multiply the base */
    for(i := 0; i < exp; i := i + 1)
    {
      res := self.mul(res, base);
    }
    return res;
  }
  service rand() {
    return randGen.getRandVal(101);
  }
}
```

LISTING 3.4 : The `Calc` descriptor. The `self` is an internal provided port referencing the current context (it resemble `this` in Java.)

```
c := Calc.new();
connect randGen@c to default@(SomeRandomGenerator.new());
c.add(c.rand(),1);
c.mul(3,c.pow(2,3));
```

LISTING 3.5 : Using an instance (a component) of the `Calc` descriptor. The invocations of the `add`, `mul`, `pow` and `rand` services are made through the `default` port of the component (*see* Definition 10 and Section 3.3.1)

Choice 11 *A required port can be connected to a single provided port by a* regular connection *if their interfaces are compatible.*

Connections allow for the invocation of required services through a port leading to the execution of provided services via another port. A required port can be connected to a single port provided. In COMPO, connections are oriented in the sense of service invocations. Invocations are sent from required ports to provided ports, while computation results (return values) are sent from provided ports to required ports. Currently, the contracts defined by the interfaces of ports are verified only on the syntactic basis and compatibility between interfaces is based on set inclusion as indicated by the choice 8. Figure 3.6 shows a connection between the port required `randGen` of an instance of the CALC (*see* Listing 3.4) and the provided port `default` of a random numbers generator component. The graphical notation is based on the UML notation [OMG, 2011b].

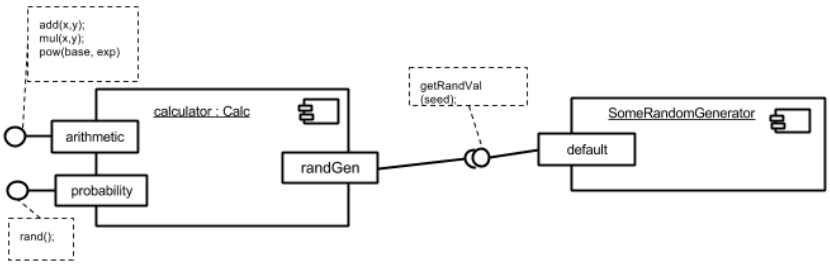


Figure 3.6 : A connection example, the connection we created by the **`connect`** statement in the second line of Listing 3.5

Until now, we have been talking about connections from required to provided ports, *i.e.* ports having different roles. These connections are sometimes called "*assembly connections*" in the literature, for example [OMG, 2011b]. When the connections have *first-class* status, then the literature talks about "*assembling connectors*". From time to time, software architects need to bind port of the same role. For example, when the facade pattern [Arnout, 2004] is used in the component-based context, then it is needed to pass invocations from provided ports of a `facade` component to provided ports of its internal components. To bind ports of the same role, *i.e.* provided to provided or required to required, the "*delegation connections*" [OMG, 2011b] link the external contract of a component (as specified by its ports) to the realization of that behavior. Delegation connections represent the forwarding of invocations. A delegation connection is a declaration that behavior that is available on a component instance is not actually realized by that component itself, but by its internal components Delegation can be used to model the hierarchical decomposition of behavior, where services provided by a component may ultimately be realized by one that is nested multiple levels deep within it. Required-to-required delegation can be used to export requirements of an internal component of a component to required ports of the component.

Choice 12 *A port can be delegated to another port with the same role by a* delegation connection *if their interfaces are compatible.*

In COMPO, a regular (resp. delegation) connection is specified by the expression as follows: `(connect | delegate) <port-address> to <port-address>`.

Definition 12 (Port-address) *The port-address expression is written in the following form* `<port-name-A>@<port-name-B>` *and returns a reference to the* `<port-name-A>` *port of a component connected (the* target*) to the* `<port-name-B>` *port. If, the* `<port-name-B>` *port is the* `self` *port then the* `<port-name-A>` *port can be either external or internal else the* `<port-name-A>` *port can only be a name of an external port.*

For example the last connection `connect backEnd@fE to default@bE;` from the architecture section of Listing 3.1 connects the `backEnd` port (of the component connected to the `fE` internal required port) to the `default` port (of the component connected to the `bE` internal required port.)

The `<port-name-B>` part of the port-address expression can be substituted by any expression returning a port. For example the following port-address expression `arithmetic@(Calc.new())` returns the `arithmetic` port of a newly created instance of the CALC descriptor, where the instantiation mechanism (*see* Section 3.3.1) returns the default port of the created instance.

In case of collection ports it is sometimes needed to address a specific port of a collection. In such case, the `<port-name-A>` and `<port-name-B>` parts of the *port-address* expression can be written with an integer index specified within brackets. For example the following port-address expression `default@regHa[2]` returns the `default` port of the second component connected to the collection port `reqHa`. An error occurs when brackets are used for a single (not collection) port and when the specified index does not exist.

With COMPO we address the problem of capturing dynamic architectures. When an architecture changes in the time, *e.g.* a component is substituted with another compatible component, it is often needed to *disconnect* a connection between components. Therefore we provide the *disconnect* statement. The statement is specified by the template as follows: `disconnect <port-address> from <port-address>` and it simply removes the connection between the specified ports. As we will see in Chapter on inheritance (Chapter 4), the disconnect statement can be used for specialization of inherited architectures.

3.3 Mechanisms

Similarly to the previous section, here we present in detail the core component-based mechanisms. For each mechanism we present the general motivation, definitions and design choices made for COMPO in order to provide a suitable solution.

3.3.1 Component instantiation

An instance, in object-oriented programming, is a specific realization of any object. An object may be varied in a number of ways and each realized variation of that object is an instance. The creation of an instance is called *instantiation*. In languages that create objects from classes, an object is an instantiation of a class. That is, it is a member of a given class that has specified values rather

than variables. Similarly, instantiation in descriptor based COLs is a mechanism for building new components according to the specification a descriptor defines. Such components are then called instances of the descriptor. For example in ComponentJ [Seco *et al.*, 2008], instances (called "*objects*" in ComponentJ's terminology) are created by applying the instantiation operator `new` on a component definition (called "*component*" in ComponentJ's terminology.)

In general, the majority of current programming languages use a form of the instantiation operator. As pointed by [Cointe, 1987] the instantiation mechanism has two phases: to allocate a memory for the new instance and to give an initial value to each instance slot[6] described in the descriptor of the instance.

In COMPO, descriptors define the structure of components (their instances). In the allocation phase of the instantiation mechanism, we analyze descriptor's external and internal contract, *i.e.* the ports it defines, and for each port the mechanism allocates a memory space. The structure and the amount of the memory needed for each port depends on a COMPO's interpreter implementation. The initialization phase happens in two steps. During the first step we set the references associating each port with its corresponding port description. These references can later be used to query the interface, the name, etc. of each port. The second step works with the architecture section of descriptors which describe connections between ports of the created component and ports of internal components. We process each connection description, *i.e.* evaluate both port-address expressions and then we set the binding reference between ports. The above lines are captured in the Algorithm 1.

Algorithm 1: Instantiation pseudo-algorithm

Data:
  d : a descriptor
forall the *pd ports declarations in d* **do**
  allocate memory space for a new port;
  set reference from the port to *pd*
forall the *c connection declarations in d* **do**
  process c

```
class X {
  private Point p;
  public X(Point x) { p = x }
}
Point mp = new Point();
X x = new X(mp)
```

LISTING 3.6 : Breaking encapsulation with parameterized constructor in Java. After the last line was executed, the `mp` reference should be invalid, otherwise someone has a reference to the object which should be private for the new instance of `X`

In fact, the architecture section of descriptors resemble the constructors in the object-oriented world. The resemblance raises new questions: *Should we allow for multiple architecture sections*

[6] Here we follow the UML terminology, where slots realize class attributes

within a descriptor? and *Should be the architecture section parameterized?* In fact, the answer to the second question answers the first question, because the usual practice is to use overloading [Meyer, 2001 ; Beugnard et Sadou, 2007] and dynamic dispatch [Lippman, 1996] to distinguish multiple constructors by their parameters count and types. The answer for the second question is related to the encapsulation of instances. Consider the Java example in Listing 3.6 where the `x` argument passed to the constructor of `X` is assigned to the private slot of the new object. Hence the encapsulation of the new object is broken because someone else is able to manipulate with the object (via the `mp` reference) that should be private for the instance of `X`. The encapsulation could be preserved by invoking the `X`'s constructor with the following expression: `X x = new X(new Point()).` Unfortunately it is not easy to ensure that such constructors will be always invoked in the same fashion. Therefore, in the current version of COMPO, we have made the following choice:

Choice 13 *The architecture section of descriptors could not be parameterized.*

3.3.2 Service invocation

The service invocation mechanism requires the following data:

- a port through which the invocation is sent,
- a name of the service to be invoked (similar to *selector* in the object world)
- a set of arguments

For example: `aPort.selector(arg1, arg2)`

The emitting port The service invocations are made in the source code of services . Syntactically, a service invocation is similar to sending a message but it is done through a port called emitting port. On other words, service invocations are not sent to the actual implementor of the behavior (as it is in OOP, for example) but they are send to the ports.

Choice 14 *A service invocation is always made via a port of a component.*

The Choice 14 is motivated by the need to ensure the *communication integrity* in software architectures [Luckham *et al.*, 1995b ; Aldrich, 2003] which say that any communication between components must happen through a well described connection.

Definition 13 (Communication integrity) *Each component in the implementation may only communicate directly with the components to which it is connected in the architecture*

Because connections are established between the ports, it is imposed that the service invocations are also performed via ports making explicit all the dependencies they induce. Indeed, it was possible to directly invoke a service of component from another component, but it would introduce a "hidden

dependency in the code" between these two components which is not described by any connection. In addition, for invocations made through a required port, as shown in Figure 3.7, the receiver is unknown for the `pm` component in the implementation and therefore it cannot be referenced directly. When one assemble an application, he select and connect the receiver component through ports. Invocations of services through a required port then cause the execution of services of the component that receives them.

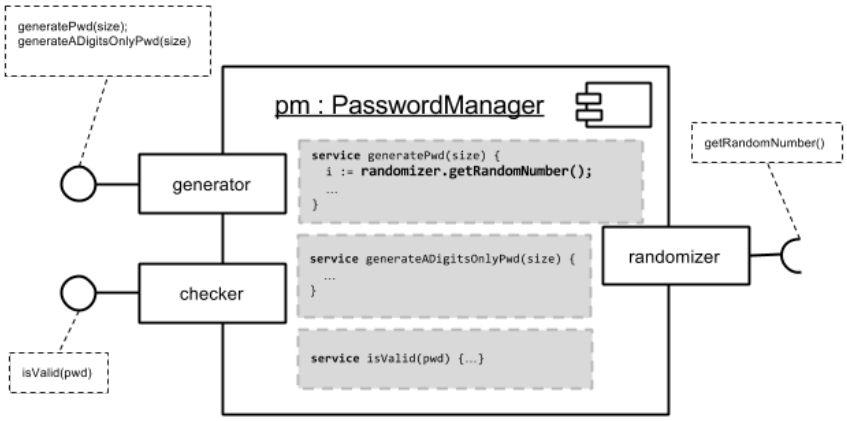


Figure 3.7 : Invocation of the required service `getRandomNumber` made through the port `randomizer` of component `pm`

Invocations via the required ports allow for better decoupling since the establishment or removal of connections permits to set the receiving component which then effectively treats the service invocations. In this context another question raises: *Do we need to invoke services through provided ports?* Unlike invocations made through required ports, provided ports do not promote decoupling as the emitting port referenced in the code is a provided port and therefore it belongs to the receiver component that actually process the invocation. Although they do not promote decoupling and fix receiver's component in the code, the service invocations made through provided ports seems to be necessary for two reasons: (i) to invoke internal services and (ii) to invoke services provided by internal components of a composite.

Because service invocations are always made through a port and, in the same time, there can be services which are not provided through any port (internal services), how is it possible to invoke an internal service? Is surprising, that this problem is addressed weakly by existing COLs. In Julia and ArchJava the components are implemented by a Java class and therefore it is possible to invoke an internal service by sending a message using the pseudo-variable `this` . In COMPO, we make the following choices:

Choice 15 *Any component possesses an internal provided port named* `self` *through which all services defined in the component are available.*

The `self` port provide an integrated and uniform solution for the invocation of internal services. Any component has the internal `self` port and this is why we usually omit it in our figures .

Parameters The parameters of services and arguments passing raises many questions when invoking services: What is a parameter? Do we really need parameters or would it be possible to use connections instead? If yes, what happen when we pass arguments? These questions were well addressed in SCL and therefore we take over the following choices made for SCL:

Choice 16 *The parameters of the service invocations are references to ports .*

The Choice 16 answers the initial question because it determines the nature of the parameters. It helps to overcome the problem of violating the communication integrity [Léger *et al.*, 2006], because service invocations sent to arguments are made through ports. If the parameters are references to ports, then what is the difference between a parameter and a required port? SCL's answer is: The difference between the parameters of the required services and ports is, apart from the syntax, the *scope* of the identifier and its lifetime (*extent*). Parameter's scope is the context of a single service, while the scope of a required port is the context of a component. Moreover, parameters live only for the time of the service's execution, but required ports exist as long as the component that owns them exist. SCL and COMPO integrate the following solution:

Choice 17 *Arguments passing is made by the automatic establishment and removal of connections. Any component has a required collection port named* `args`*. During a service invocation, the arguments, i.e. ports,* $\langle a_1, a_2, ..., a_n \rangle$ *are each respectively connected to* $\langle \mathtt{args}[1], \mathtt{args}[2], ..., \mathtt{args}[n] \rangle$*. The identifiers of the parameters are actually alias identifiers of ports* `args` *. At the end of the execution of the service, all connections to* `args` *ports are removed. In the case when a service invocation is delegated, because the receiving port is delegated, the appropriate delegation connection for the* `args` *port is also automatically established and removed.*

This choice propose a uniform mechanism respecting the communication integrity since it is impossible for components to communicate otherwise than though connected ports.

Figure 3.8 shows an example of using this service invocation mechanism processing steps as follows:

1. Emitting a service invocation through a port. In Figure 3.8 component `calculator` emits through its port `randGen` a service invocation for service `getRandVal` with a sole argument being a reference to the `default` port of an instance of some SEED descriptor.
2. The receiver (component `gen` in our example) receives and processes the invocation. We will return to the way a component processes a service invocation later in this Section.
3. The ports passed as an argument to the invocation are connected to the `args` ports of the receiver component (*see* Figure 3.8).
4. Then the service is executed. A mechanism for transparent *aliasing* (managed by an COMPO's interpret) allows the programmer to use names specified for the parameters in the implementation and does not require him to use the `args` ports directly. In our example, the source code of the service `getRandVal` of `gen` component uses the identifier `seed` as parameter name and not `args[1]`.

5. At the end of the execution of the service, all connections to the `args` ports are removed.

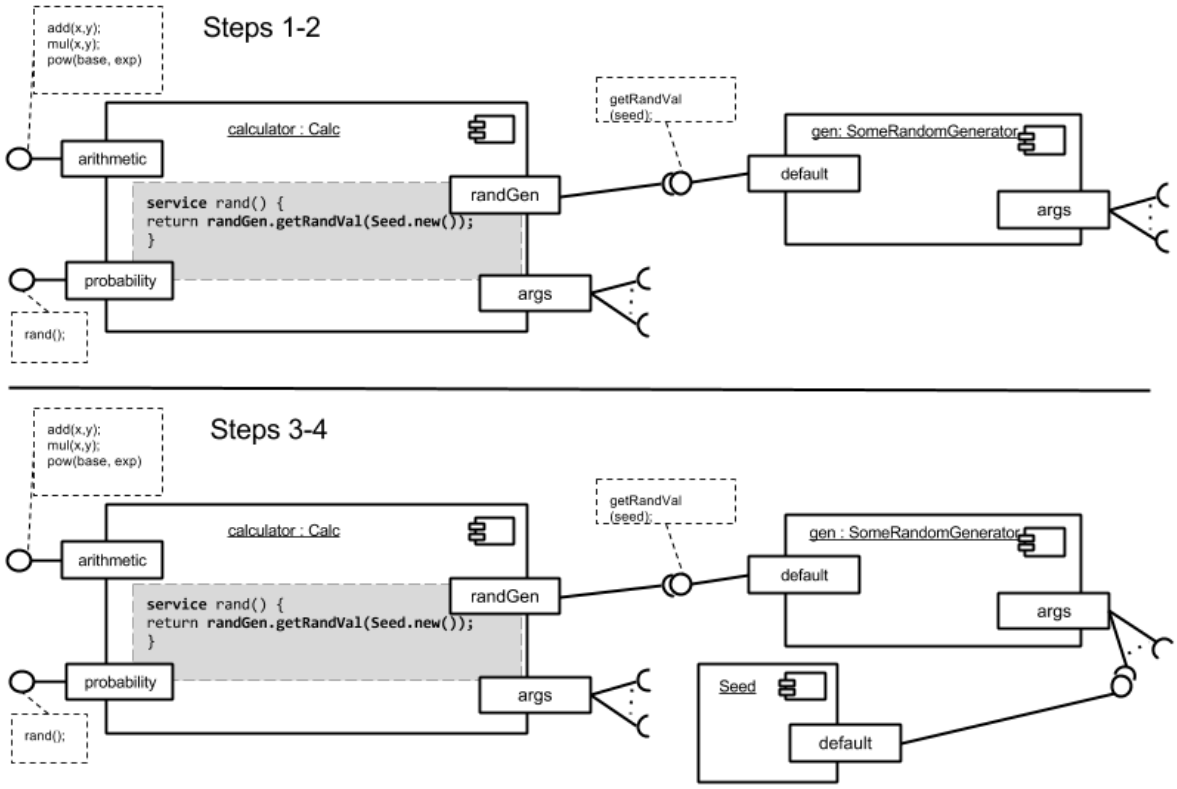


Figure 3.8 : Illustration of service invocations treatment in COMPO

There are two last questions to be answered: *How return values are treated?* and *How one can store an argument of a service invocation or a return value?* When a service `foo` invokes (through a port) another service `bar` and `bar` returns a value, the value is (in the code of `foo`) either passed as an argument for another service invocation (then it is treated as usual) or the value is about to be stored[7]. First, it is important to note that a value is always a reference to a provided port. In the case of arguments an argument value is the reference to a provided port which is connected to the appropriate `args` port. In the case of return value the following code snippet show four basic cases:

```
return 1;              /* Case 1 */
return Calc.new();     /* Case 2 */
return providedPort;   /* Case 3 */
return requiredPort;   /* Case 4 */
```

As we will see in the chapter about reflection (*cf.* Chapter 5), numbers, strings, symbols, etc.... are also components, thus Case 1 actually means that a reference to the `default` port of component

[7] Indeed, there is the third option that nothing is done, in that case the return value is simply destroyed.

representing number 1 is returned. Case 2 is an example of instantiation, as we will in the section about instantiation mechanism (*cf.* Section 3.3.1), the mechanism returns a reference to the `default` port of the new instance. Case 3 already returns a reference to a provided port. In Case 4, we return a reference to the provided port to which the required port is connected. Or there is an error if the required port is not connected.

There are two options for storing or referencing a value in COMPO: connections (regular or delegation) and assignment operator `:=`. Both options may lead to many dangerous situations (*see* Listing 3.7) when internals of the component owning the port representing return or argument value might be exposed, referenced and potentially abused. Therefore to preserve the encapsulation of components, we make the following two choices:

Choice 18 *It is forbidden to use, in the code of a service, a value of a service parameter (argument) to build a regular or delegation connection.*

Choice 19 *Assignment operator always stores a clone of the value being on the left side of the assignment expression.*

```
/* suppose existence of:
   - an internal required port 'irp'
   - an internal provided port 'ipp'
   suppose that:
   - invocation x.bar() returns a reference to the provided port
                        of an internal component of the component connected to x
*/
service foo(x) {
  |temp|
  /* connect 'irp' to the default port of a component connected to x */
  connnect irp to default@x;

  /* connect required port reqPort of a component connected to x to the 'ipp' */
  connnect reqPort@x to ipp;

  /* connect 'irp' to the default port of an internal
     component of the component connected to x */
  connect irp to default@(x.bar());

  /* storing a reference to a foreign internal component */
  temp := x.bar();
}
```

LISTING 3.7 : Dangerous behavior when referencing or storing return values and invocation arguments.

In the following we define two basic algorithms of the service invocation mechanism in COMPO. The two algorithms concern:

- emitting a service invocation through a required port
- receiving a service invocation through a provided port

Algorithm 2 captures the treatment of a service invocation in a required port. This algorithm consists of three cases. The first (*see* Fgure 3.9 case 1) corresponds to a regular connection between a required port and a provided port. The invocation in this case is forwarded to the specified connected provided port and treated according to the algorithm 3 (described below after). The second case (*see* Fgure 3.9 case 2) corresponds to a delegation connection. The invocation is then transmitted to the delegated required port which treats the invocation by the same algorithm 2. The third and last case corresponds to a unsatisfied dependency when the required port is not connected at all. Such case produces an error. The treatment of this error should result in an exception being thrown. However, this is beyond the scope of this thesis because it would require to introduce a exception handling system for COMPO which should be adapted to the component-based context at it has been studied in [Souchon, 2005].

Algorithm 2: Sending a service invocation through a required port

```
Data:
    i : a service invocation
    r1 : the required port through which i is emitted
if r1 is connected to a provided port p2 then                    /* case 1 */
    transmit i through p2;
else
    if r1 is delegated to a required port r2 then                /* case 2 */
        delegate i through r2;
    else                                                  /* r1 not connected */
        error case;
```

Algorithm 3 captures the treatment of a service invocation in a provided port. This algorithm consists of four cases. The first (*see* Fgure 3.10 case 1) is to delegate the service invocation to another provided port that deals with it by the same algorithm. This happens only if the requested service is not implemented by the current component. In the second case (*see* Fgure 3.10 case 2), the requested service is performed as defined by the component to which the receiver port belongs. In the third case (*see* Fgure 3.10 case 3), the component to which the port belongs does not implement the requested service and so the lookup mechanism specified in Chapter 4 has to be executed. The last case is an error case where the receiving port is not connect, nor delegated and the demanded service is not implemented by the current component.

These algorithms have been designed to take into account issues or cases requiring special attention, such as the one shown in Figure 3.11. Where a service invocation is transmitted through the port `r1` of the internal component `c1` in the code of the service `foo` . The treatment of this invocation according to algorithm 2 resulted in error. There is not a rule to automatically run the service `bar` of the composite. Such a rule would be problematic when providing various services for two internal components that require a service named `bar` . In our example, the service invocation of `bar` service

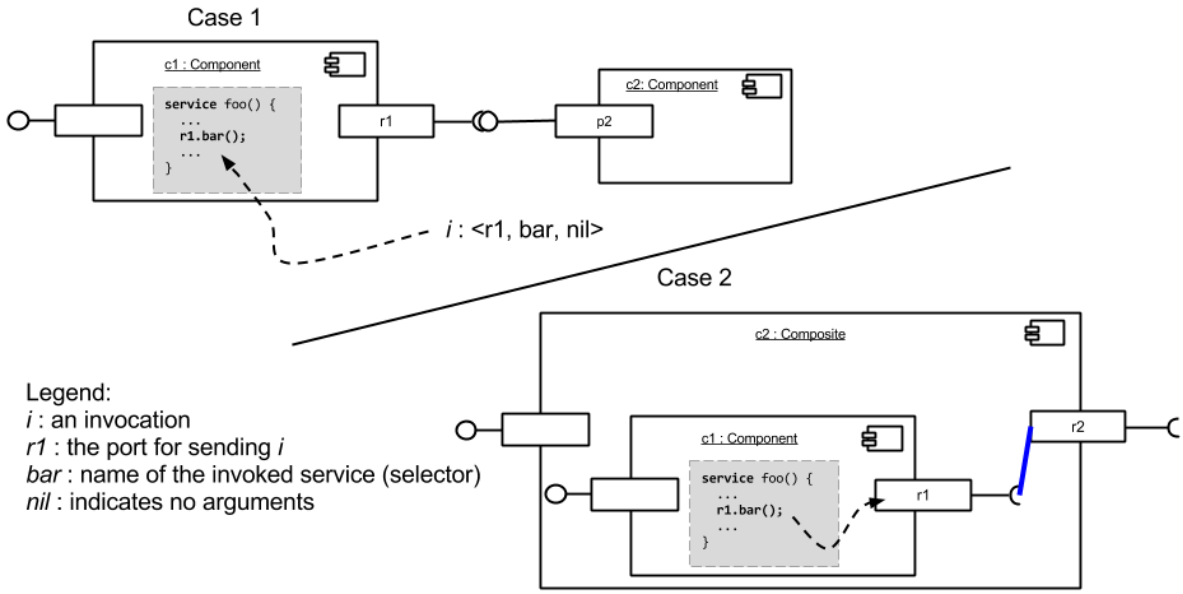


Figure 3.9 : The basic cases concerning service invocations through required ports

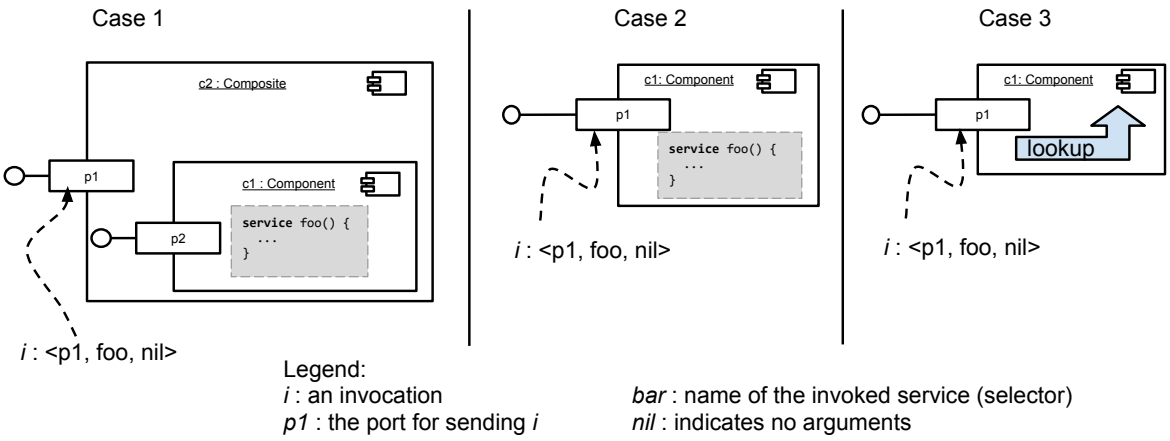


Figure 3.10 : The basic cases concerning service invocations through provided ports

through `r1` in component lead `c1` to the execution of the service `bar` of composite `c2`, the architect must establish a connection between the port `r1` of the internal component `c1` and port `self` of the composite `c2`.

Algorithm 3: Receiving a service invocation via a provided port

```
Data:
    i : a service invocation
    p1 : the provided port through which i was received
if selector(i)∈ interface(p1) then
    if p1 is delegated to a provided port p2 then                    /* case 1 */
        delegate i through p2;
    else
        if descriptor(component(p1)) possesses the service demanded in i then   /* case 2 */
            execute s
        else
            if descriptor(component(p1)) does not implement
            the service demanded in i then                           /* case 3 */
                lookup s
            else                                                     /* p1 not connected */
                error case;
else                                        /* service is not listed in the interface */
    error case;
```

3.3.3 Composition mechanism

The composition mechanism represent one of the core ideas behind components. It permits users to create a new component by connecting off-the-shelf components within the context of the new component, *i.e.* to achieve ("*development by reuse*" [Fabresse *et al.*, 2012]). These new components are then called *composite* because they are themselves made of more elementary components called *internal components*. Internal components are sometimes referenced as sub-components in the literature. We chose to not use the sub-component terminology, because it might confuse COMPO's users, in case they use inheritance to define sub-descriptors (*see* Chapter 4.) Composition is comparable to the composition relationship between UML classes.

Naturally, the first question which comes to one's mind is: *Do we need composites?* To answer this question, we recall the decoupling aspect of required ports which increases the potential for reuse. This technique for extracting dependencies from the source code in the form of required services is sometimes called *factoring out* [Seco et Caires, 2000]. In the example of Figure 3.6, an instance of CALC can be used with any component providing a service for generating random numbers. Although this *factoring out* technique allows for a better decoupling, it also poses problems of transition to the scale and reuse. Indeed, it is currently impossible to directly reuse an assembly of components, *e.g.* an instance of CALC connected to an instance of SOMERANDOMGENERATOR Figure 3.6. This means that for every application where the architect wants to integrate an instance of CALC, it must include an instance of SOMERANDOMGENERATOR again and establish the necessary connection between these two components. Composites include a response to these needs, namely:

- encapsulate an assembly of several components to hide certain details in a software architec-

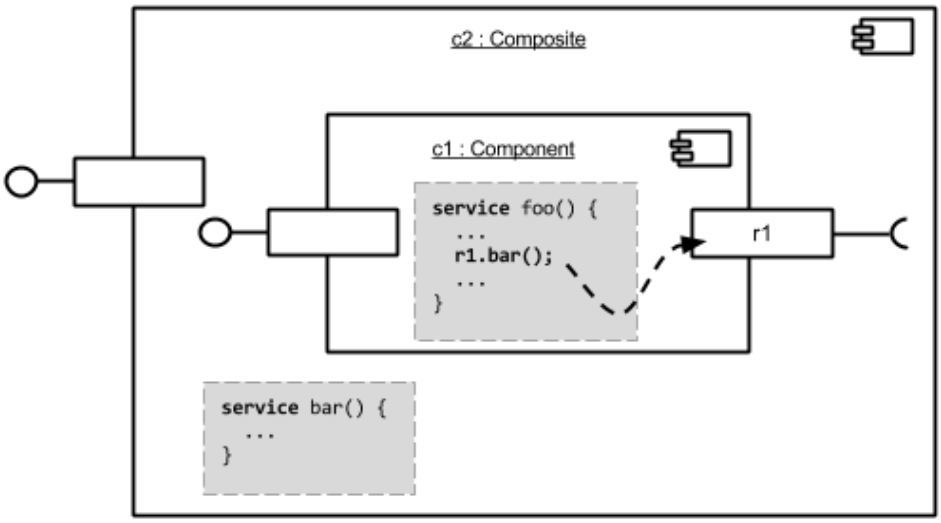


Figure 3.11 : An example of a problematic case of service invocations

ture,

- directly reuse assemblies of components.

The ADLs were the forerunners in providing the concept of *configuration*. In WRIGHT for example, *configuration* is a set of components connected through connectors. Unlike a configuration, a composite is component. The models such as Fractal and ArchJava propose the concept of *composite* to represent assemblies of components. These models are called *hierarchical* as a composite can be decomposed into a collection of interconnected *sub-components*, each sub-component may be a composite or a simple component. We chose to incorporate a similar approach.

Choice 20 *A* composite *is a component having one or more internal components.*

Considering composites as components (*cf.* Choice 20) allows :

- to put on the shelf and reuse;
- to create partially configurable architectures by use of required ports;
- to make them more understandable, since complex architectures can be examined at different levels of granularity depending on whether or not we detailed the content of a composite.

Before going further, it seems necessary to present a problem — often overlooked in existing approaches — regarding the design of composites requiring to not apply the *factoring out* principle. Indeed, when a programmer develops a composite, it selects and sets the internal components it uses.

This necessarily leads to a coupling between a composite and its internal components. The design of a composite thus requires a choice between what needs to be outsourced (via required ports) to keep a high potential for reuse and what needs to be made using internal components to hide details. The systematic use of composites without outsourcing does not define reusable components. However, it is impossible to prevent this in a COL. We can limit the adverse effect of the composition by requiring that each descriptor is defined separately (*cf.* choice 21).

Choice 21 *A composite does not contain the descriptors of its internal components, but possess references to them.*

By this Choice 21, we prohibit nesting of descriptors (the parallel in Java would be to prohibit *inner classes*). Our goal is that all descriptors are placed on the shelf to be reused. This choice is motivated by the fact that a component should not just be seen as an encapsulated set of internal components but as a composite component to be putted on the shelf and possibly used as an internal component in another context. Figure 3.12 empathizes the difference between an assembly of components and a composite.

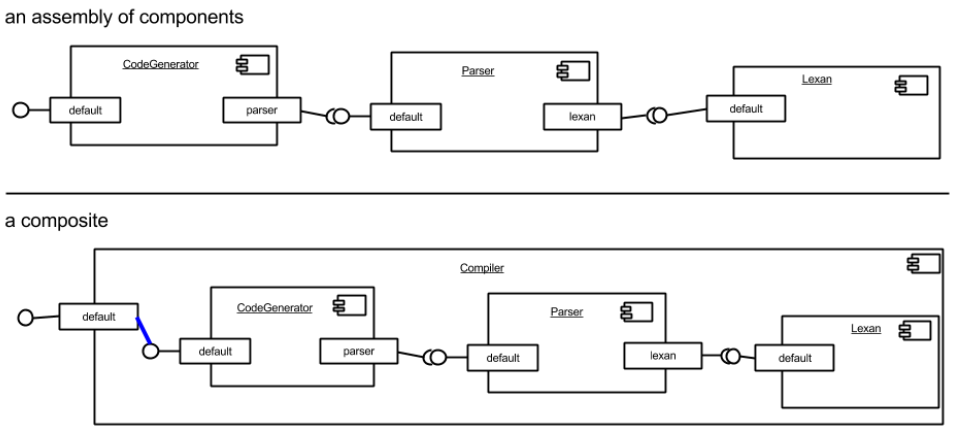


Figure 3.12 : Empathizing the difference between an assembly of components and a composite. The COMPILER can be easily putted on the shelf and reused later.

To preserve the communication integrity [Léger *et al.*, 2006], *i.e.* the fact that components communicate uniquely via ports and each communication channel has to be described by a connection, we have to answer the following question: *How should we communicate with internal components?* It is surprising that this question is not addressed by existing COLs. For example, the communication integrity is strictly abide in ArchJava, but there is no description of a communication channel (a connection) between a composite and its internal component. Internal required ports are the COMPO's answer to this question. The architect of a composite has to define an internal required port (in a descriptor of the composite) for every internal component he/she wants to reference.

Choice 22 *A composite communicates with its internal components through internal required ports, one per an internal component.*

The Choice 21 preserves the communication integrity and simplifies COMPO's design because there now only one communication protocol.

3.3.4 Substitution mechanism

One of the main difficulties of software evolution is that all artifacts produced and used during the entire software life-cycle are subject to changes, ranging from early requirements over analysis and design documents, to source code and executable code [Mens, 2008]. Updating the executable code is the last step to reflect the changes required by the new requirements. The substitution mechanism focus only on this last step: how to change a component in the running system.

Run-time change seems to the stimulus for many component-based approaches, as for example Kevoree, ArchJava, OpenCOM, Fractal, Darwin, WRIGHT, SOFA, ACME, MetaORB, DynamicTAO, ... Besides, some approaches provide languages for dynamic update description like Fractal's FScript or SOFA's DCUP. It is surprising that instead of the usual heterogeneity in terminology, etc., there is a certain consensus in the case of component's substitution. The widely accepted criterion to determine whenever a component can or cannot be change to another component in an architecture is that these two components has to be substitutable, *i.e.* their external contracts have to be compatible. For example ACOEL, ArchJava and CompJava define the subtitutability constraint and sub-type relation as defined by Liskov substitution principle:

> "*In a computer program, if S is a sub-type of T, then objects of type T may be replaced with objects of type S (i.e., objects of type S may be substituted for objects of type T) without altering any of the desirable properties of that program (correctness, task performed, etc.)*" [Liskov et Zilles, 1974].

In general, a component type is a sub-type of another one if it provides at least the same and requires at most the same. One solution to avoid the problem is to forbid substitutability [**?**]. However substitutability is important; most of reuse design patterns [Gamma *et al.*, 1995b] use it and more globally it the mechanism represent the heart of many frameworks and of the product line technologies. With respect to others, we consider the choice to constraint substitutions with such sub-type relation as a unnecessary limitation. In stead of saying that a component `a` is substitutable with a component `b` if `b` provides at least the same and requires at most the same as `a`, we chose the following:

Definition 14 (Substitutability) *A component `a` is substitutable with a component `b` if the descriptor of `a` is compatible with the descriptor of `a` and all requirements of component `b` will be satisfied after the substitution.*

Such a non-restrictive system delegates responsibility for requirements satisfaction to the users. To define the compatibility relation between descriptors the external contract is used. An *external contract* is defined by a descriptor and computed as a set of tuples, each tuple being a triplet `portname-interface-role`. For example, the contract for descriptor CALC from Listing 3.4 is:

```
{
   {arithmetic-{add(x, y); mul(x, y); pow(base, exp)}-provides},
   {probability-{rand()}-provides},
   {randGen-{getRandVal(seed)}-requires}
}
```

A contract cannot be computed as a set of all provided and required interfaces, because information about interfaces' roles (provided or required) is missing [Seco et Caires, 2000]. For example a component providing the service `compile` and a component with a required service `compile` will be then indistinguishable. Also port structure has to be respected, component providing a `run` service via a port `a` and a component providing a service `run` via a port `B`, are structurally different.

External contracts permit us to define compatibility relationship between descriptors:

Definition 15 (Descriptors' compatibility) *Component descriptor* A *is compatible with descriptor* B *if for each tuple of the external contract of* A *there is at least one unique* ***unused*** *tuple in the external contract of* B *having a compatible interface and the same role.*

Very important is the word "*unused*" in the Definiton 15, it empathizes the fact that once a tuple of the external contract of B is matched with a tuple of the external contract of A, it cannot be matched again. The interface compatibility were already defined in Choice 8.

Every time when a user tries to substitute components, he/she uses the *replace routine* implementing the substitution mechanism. The replace routine, in first step, checks if a descriptor of the new component is compatible with descriptor of the original component and, in second step, if all requirements of a new component will be satisfied after the substitution. If everything is OK, the routine performs the substitution, *i.e.* is disconnects the original component from an architecture and connects the new component to the architecture. The replace routine performs each substitution as an *atomic operation*, which means that no service invocation can be emitted or received during this operation. If the routine fails an error occurs and the substitution is not performed. The syntax and semantics of the replace routine is as follows: **`replace`** `<portA>` **`with`** `<portB>`. Where `<portA>` is a port to which the original component (that should be replaced) is connected. `<portB>` is a port to which the new component (that should replace the original component) is connected.

Figure 3.13 shows an example where the replace routine is used to substitute an instance of the CALC (defined in Listing 3.4) with an instance of the EXTCALC (defined in Listing 3.8).

```
Descriptor ExtCalc {
  provides {
    arithmetic : { add(x, y); mul(x, y); pow(base, exp) };
    probability : { rand() };
    combinatoric : { fib(x); }
  }
  requires {
    randGen : { getRandVal(seed); };
    stack : { push(val); pop(); empty(); }
  }
  service add(x, y) { return x + y }
  service mul(x, y) { return x * y }
  service pow(base, exp) {
    | res i |
    res := 1;
    /* the exp times multiply the base */
    for(i := 0; i < exp; i := i + 1)
    {
      res := self.mul(res, base);
    }
    return res;
  }
  service rand() {
    return randGen.getRandVal(101);
  }
  service fib(x) {
    | tot a |
    tot := 0 ;
    while(stack.empty() == false) {
      a := stack.pop()
      if(a < 1) { tot := tot + 1; }
      else {
         stack.push(a - 1);
         stack.push(a - 2);
      }
    }
    return tot;
  }
}
```

LISTING 3.8 : The EXTCALC descriptor.

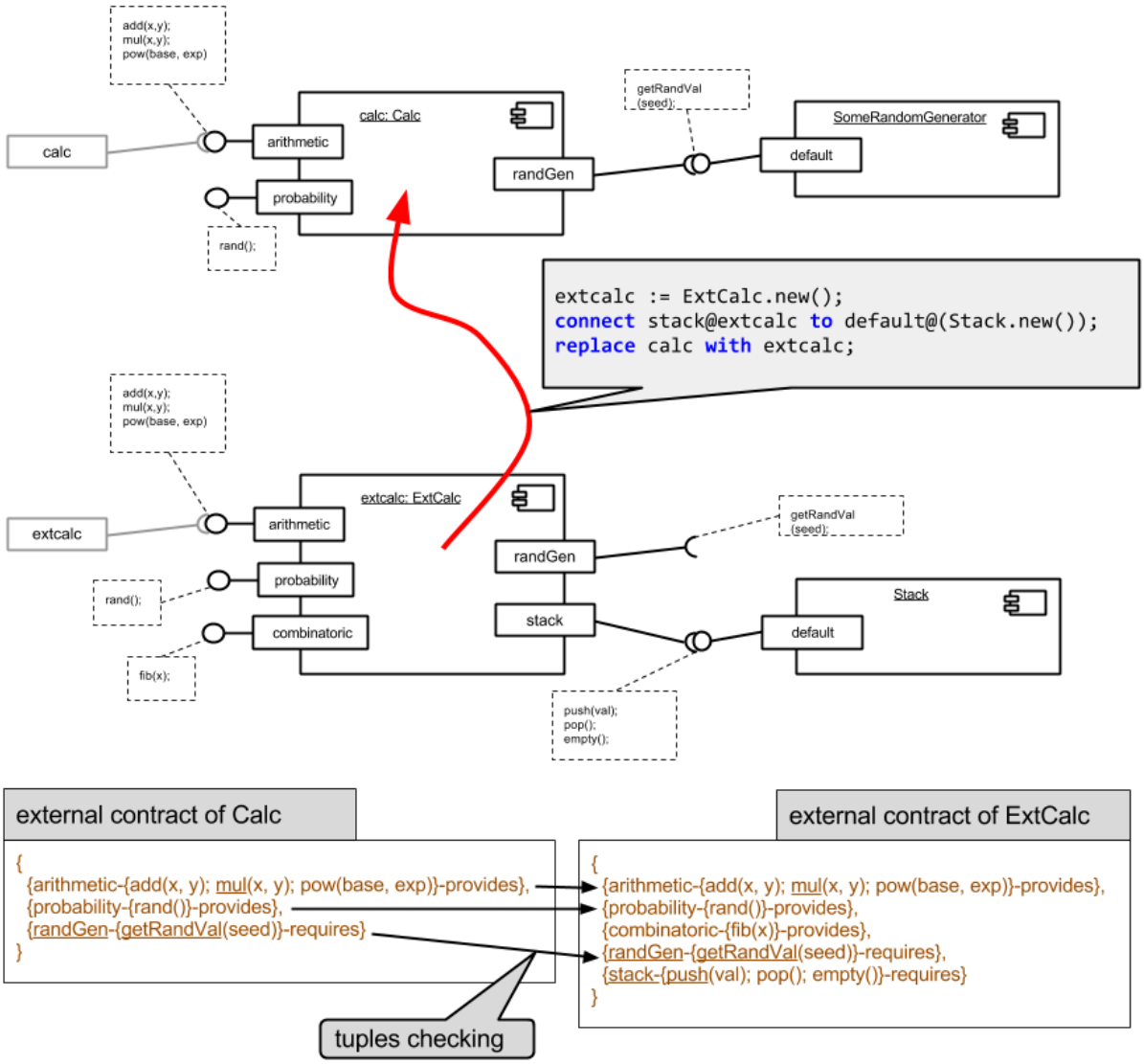


Figure 3.13 : An example of a substitution. The *replace routine* is used to substitute an instance of the CALC (defined in Listing 3.4) with an instance of the EXTCALC (defined in Listing 3.8.) The compatibility of the descriptors is illustrated by the tuples checking in the bottom of the figure.

3.4 Recapitulation

In this section we repeat the definitions and the choices that we have made in this chapter.

3.4.1 Definitions

Component-oriented language (COL) A language used to design and implement software components (*development for reuse*) with well defined external contracts; that can be stored in libraries (also called components on shelves) and, in the same time, to develop applications by assembling off-the-shelve software components, that is, to allow to describe software architectures in terms of connecting components selected from libraries (*development by reuse.*) (*cf.* 1)

The architecture of a component The architecture of a component is a description of an internal composition, *i.e.* a system of internal components and their inter-connections, according to which the component will be initialized. (*cf.* 2)

COMPO-descriptor A descriptor defines the *structure* and *behavior* of its instances called components. The behavior is given as a set of services definitions. The structure is given by description of ports and by description of the architecture. Descriptions of external (resp. internal) ports define an *external contract* (resp. an *internal contract*) of instances of the descriptor. (*cf.* 3)

Required (resp. provided) port Required (resp. provided) port of a component is a named connection and communication point through which the component requires (resp. provides) a set of services. (*cf.* 4)

Universal interface $*$ In case of provided ports, the universal interface $*$ means that a port offers all services already provided by the descriptor of a component that owns the port. In case of required ports, it means that any service could be invoked through such a port. (*cf.* 5)

Collection port A named and ordered collection of required or provided ports. Each port of the collection can be accessed by an index. (*cf.* 6)

The `sizeof` operator The `sizeof` operator, when applied on a port, returns the count of connections the port participate in. (*cf.* 7)

Provided service a functionality defined in the source code of a component, which is offered to other components through ports, so they can invoke it later. (*cf.* 8)

Internal service a service which is not offered trough any provided port of the component (that defines it) is not accessible from outside of the component. (*cf.* 9)

Required service a service which is necessary for implementation of the behavior of a component `X` (invoked in the code of its services), provided by an external component connected to the required port of the component `X`. (*cf.* 10)

Service signature Each service has a signature given by the following template: `<selector> (<parameter1-name>, <parameter2-name>, ... )`. Two service signatures are compatible if they have the same selector and the same count of parameters. (*cf.* 11)

Port-address The port-address expression is written in the following form `<port-name-A>@<port-name-B>` and returns a reference to the `<port-name-A>` port of a component connected (the *target*) to the `<port-name-B>` port. If, the `<port-name-B>` port

is the `self` port then the `<port-name-A>` port can be either external or internal else the `<port-name-A>` port can only be a name of an external port. (*cf.* 12)

Communication integrity Each component in the implementation may only communicate directly with the components to which it is connected in the architecture (*cf.* 13)

Substitutability A component `a` is substitutable with a component `b` if the descriptor of `a` is compatible with the descriptor of `a` and all requirements of component `b` will be satisfied after the substitution. (*cf.* 14)

Descriptors' compatibility Component descriptor A is compatible with descriptor B if for each tuple of the external contract of A there is at least one unique **unused** tuple in the external contract of B having a compatible interface and the same role. (*cf.* 15)

3.4.2 Choices

Choice 1 A *component* is a run-time entity, instance of a *descriptor*, which provides and requires services through ports.

Choice 2 The component descriptors are placed on the shelf.

Choice 3 A component (not a descriptor) is a subject for assembling.

Choice 4 Descriptors cannot be combined.

Choice 5 The description of the architecture of a component is a part of a descriptor of the component.

Choice 6 A component has *unidirectional* ports.

Choice 7 A *interface* is associated with a port.

Choice 8 A *interface* specifies a set of signatures of services. The *interface* compatibility is based on subtyping relationship between their types which is based on the inclusion of sets of signature services.

Choice 9 A component can have *collection ports*.

Choice 10 Every component has a port named `default` through which all services provided by the component are available. The instantiation mechanism of descriptors returns a reference to the `default` port of the newly created component.

Choice 11 A required port can be connected to a single provided port by a *regular connection* if their interfaces are compatible.

Choice 12 A port can be delegated to another port with the same role by a *delegation connection* if their interfaces are compatible.

Choice 13 The architecture section of descriptors could not be parameterized.

Choice 14 A service invocation is always made via a port of a component.

Choice 15 Any component possesses an internal provided port named `self` through which all services defined in the component are available.

Choice 16 The parameters of the service invocations are references to ports .

Choice 17 Arguments passing is made by the automatic establishment and removal of connections. Any component has a required collection port named `args`. During a service invocation, the arguments, *i.e.* ports, $\langle a_1, a_2, ..., a_n \rangle$ are each respectively connected to $\langle args[1], args[2], ..., args[n] \rangle$. The identifiers of the parameters are actually alias identifiers of ports `args` . At the end of the execution of the service, all connections to `args` ports are removed. In the case when a service invocation is delegated, because the receiving port is delegated, the appropriate delegation connection for the `args` port is also automatically established and removed.

Choice 18 It is forbidden to use, in the code of a service, a value of a service parameter (argument) to build a regular or delegation connection.

Choice 19 Assignment operator always stores a clone of the value being on the left side of the assignment expression.

Choice 20 A *composite* is a component having one or more internal components.

Choice 21 A composite does not contain the descriptors of its internal components, but possess references to them.

Choice 22 A composite communicates with its internal components through internal required ports, one per an internal component.

3.5 Related work

The COMPO language builds on diverse fields of related work, including architecture description languages, component frameworks, module systems or modeling tools. COMPO integrates ideas from many of these areas in order to provide a rich architecture specification language and a practical programming language.

Among the component-based approaches, only some are consistent with Definition 1, for example: ArchJava ComponentJ Piccola or Lagoona. Indeed, all these languages allow for component-based development and offer, even if called differently, the core concepts and mechanisms of the component-based approach, unlike the proposals that primarily use standard languages to design frameworks for creating component-based solutions, such as Fractal's Java implementation called Julia.

COMPO shares with SCL (the predecessor of COMPO) many features like unique communication protocol, unplanned connections support or services' arguments passing. With respect to SCL, COMPO tries to push further in modeling aspect, its explicit architectures support, meta-model and inheritance system (described in next sections) boots modeling power of the language and provide basis for Model Driven Development.

In opposite to COMPO, some COLs, like ArchJava or CLIC model are not implementation independent. For example, CLIC focus on symbiosis between CLIC and Smalltalk plus it enables to benefit from modularity and reusability of components without sacrifice performance. Compared to COMPO, modeling powers of CLIC are limited, the model allows components to have only one provided port. The authors argue that it is hard to split component functionality over multiple ports, because developers do not know beforehand, which services will be specified by each required port of client component. ArchJava is an extension of the Java language to express application architecture directly in the source code and ensure the adequacy of the architectural descriptions and implementation.

The framework family focus primarily on practical issues such as deployment, packaging or non-functional services. They are lacking some concepts of pure component models, as for example explicit internal composition description. With the CCM model, the OMG aims in building a bridge between the pragmatic problems and concepts of architecture description. Fractal is a component model which highlights the concepts of composites and sharing. SOFA focuses on the dynamic nature with the dynamic replacement of component or to the connectors that are reification of connections between components.

Except from COM, *Javabeans* and EJB approaches, the composition mechanism is a central mechanism in the most of approaches. It is based on different types of connections between ports (or interfaces for models without ports). The majority of approaches distinguishes delegation links between a composite and one of its internal components and the "normal" connections between two components. CCM also distinguishes connections types required/provided (for synchronous communication) and event connections (for asynchronous communication). ArchJava also proposed in its first the concept of service versions released (broadcast) that resembles the event links.

Verification of connections between components is, as in COMPO, based on the sub-typing relationship between the interfaces for syntax compliance. ArchJava also offers the most advanced typing system more complete control of the communication integrity between components.

In the generative family, a number of architecture description languages (ADLs) have been defined to describe, model, check, and implement software architectures. Many of these languages support sophisticated analysis and reasoning. For example, WRIGHT allows architects to specify temporal communication protocols and check properties such as deadlock freedom. SADL formalizes architectures in terms of theories, shows how generic refinement operations can be proved correct, and describes a number of flexible refinement patterns. The SADL system formalizes architectures in terms of theories, providing a framework for proving that communication integrity is maintained when refining an abstract architecture into a concrete one. However, the system did not provide automated support for enforcing communication integrity. While SADL and WRIGHT are pure design languages, other ADLs have supported implementation in a number of ways. UniCon's tools use an architectural specification to generate connector code that links components together. 2̧ provides run-time libraries in C++ and Java that implement 2̧ connectors. Darwin provides infrastructure support for implementing distributed systems specified in the Darwin. Although the code generation tools are convenient to programmers, they do not automatically enforce communication integrity. Furthermore, these tools support a limited number of built-in connector types, and developers cannot easily define connectors with custom semantics. Architectures in Rapide can be filled in with implementations in an executable sub-language or in languages such as C++ or Ada. The Rapide system

includes a tool that dynamically monitors the execution of a program, checking for communication integrity violations. The Rapide papers also suggest that integrity could be enforced statically if system implementers follow style guidelines, such as never sharing mutable data between components. However, the guideline forbidding shared data prohibits many useful programs, and the guidelines are not enforced automatically.

3.6 Summary

In this chapter, we presented the heart of this thesis, a component-oriented programming and modeling language named COMPO. We began by the philosophy of the language where we have defined the core concepts and mechanisms the component-based approach. Then in each section we have detailed the concepts and mechanisms, one by one. For each concept or mechanism we have presented COMPO's realization and design choices together with the reasons which led us to incorporate this particular solution.

Thus, we believe that COMPO offers a reasonable solution for all the core concepts and mechanisms and thus it is suitable for component-based development, which is not always the case of languages like Julia, ArchJava or ComponentJ as we have tried to show throughout the chapter. Section 3.4 is a recapitulation of the definitions and the choices that we have made for COMPO yet. However, remember the points that make the COMPO language specific:

- the service invocation mechanism and in particular the arguments passing in terms of temporary connections and preserving the integrity of communications,
- the internal port named `self` to make self-references using the standard service invocation,
- the internal required ports used for communication with internal component of a composite.

CHAPTER 4

Integrating inheritance

Every piece of knowledge must have a single, unambiguous,
authoritative representation within a system.
Don't Repeat Yourself Principle

Preamble

In this chapter, we present an original and complete inheritance system for our component-based programming and modeling language. We first motivate the need for an inheritance system by showing cases where an inheritance mechanism is inevitable for reusing the structural definition of descriptors. Then we identify the subjects for inheritance in a component-oriented programming language and for each subject we present (section by section) our solution for extending descriptors and specializing inherited subjects. Finally, this chapter ends with a summary.

4.1 Introduction: Do we need inheritance?

A LOWER development time/cost and an elimination of bugs are the main advantages and motivations for achieving *code-reuse* in software development. Despite its well-known pathologies [Taenzer *et al.*, 1989 ; Hürsch, 1994 ; Boyland et Castagna, 1996], *class-based inheritance* is and has been the essential mechanisms for code-reuse in object-oriented programming (OOP). To overcome the problems of inheritance, new code-reuse techniques[1] like mixins [Bracha et Cook, 1990], traits [Curry *et al.*, 1982] or aspects [Kiczales *et al.*, 2001] have been invented.

Inheritance is indeed not mandatory; many efficient languages do not integrate it (C for example). Inheritance also introduces some complexity in language implementation and in programmers code. However, the success of the object-oriented paradigm do demonstrate that its advantages are greater than its drawbacks, especially if we take into account the fact that better specifications and implementations continue to be developed [Ducournau, 2011]. Therefore we see inheritance as the major cornerstone of code-reuse in OOP for the tow following reasons:

1. For the ability it gives developers to organize their ideas on the base of an incremental concept classification (a list is a kind of collection, such an architecture is a kind of visitor, ...) which is itself one key of human abstraction power.

2. For the calculus model that makes it possible to not only reuse but adapt software, by executing an inherited code in a new context (the receiver environment).

Even if it has been successful in the world of objects, inheritance as a reuse mechanism has been underrated and considered to be controversial by the component community [Szyperski, 2002 ; Opluštil, 2003 ; Lahire *et al.*, 2004]. In ECOOP'96, Weck and Szyperski presented a contribution [Weck et Szyperski, 1996] entitled: "*Do we need inheritance?*". In their work, they argue in favor of composition as a code-reuse mechanism and say that inheritance should rather be used as a modeling aid to express relationships between different classes. Indeed, composition is already a present code-reuse mechanism in the component world (*see* Section 3.3.3) and introducing an inheritance mechanism seems to be redundant in this perspective. In this section we try to show that sole composition is not enough and that it is worth to integrate an inheritance mechanism into a component-oriented language.

There are also some approaches proposing *component sharing* to improve software reuse (black-box reuse type) as in Fractal [Bruneton *et al.*, 2006] or Ernie [Outhred et Potter, 1998]. The developer can specify an internal component that is accessible from multiple composite components. Although sharing may save resources, it breaks the encapsulation of the composite components, which is as dangerous as for example when two instances of different classes hold a private reference to a unique object in OOP.

It is true that inheritance causes problems ranging against the decoupling and encapsulation principles of the component-based approach. Before explaining these problems, remember that in OOP, a class has two types of customers: (i) those who instantiate (*instantiating* customer); and

[1] See for a good survey [Lahire et Quintian, 2006]

(ii) those who inherit (*inheriting* customer). These are the customers of the second category that are both powerful and problematic. Problematic, because a sub-class is a privileged customer of its super-class and thus it can access the internal details and therefore it makes the encapsulation of the super-class "fragile" [Snyder, 1987 ; Mikhajlov et Sekerinski, 1998]. Figure 4.1 illustrates this problem. Assume the existence of a class `Set` whose instances represent sets. This class owns methods `add` to add an item and `addAll` to import all the elements of another set. Suppose we want to write a class `CountingSet` inheriting class `Set` and adding the code to count elements. To do so, the programmer of the class `CountingSet` must be aware of the implementation of the two methods `add` and `addAll` defined in the class `Set`. Indeed, if the `add` method is invoked in the code of `addAll` method, the class `CountingSet` should redefine the `add` method (*see* Figure 4.1 case (a)). In opposite, if the implementation of the `addAll` method does not use the `add` method and imports items directly, the `CountingSet` must override both methods `add` and `addAll` (*see* Figure 4.1 case (b)). This example shows that in many cases, the implementation of a super-class has to be known to make a sub-class, hence, the internals are exposed.

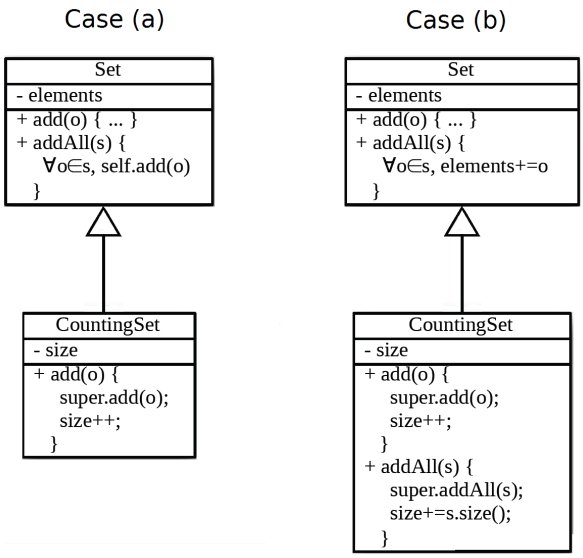


Figure 4.1 : Illustration of the fragile base class problem

The fragile base class problem shows that inheritance is in contradiction with the encapsulation and decoupling principles of the component-based approach on at least the following two points:

- it requires having access to the source code (*i.e.* to have a detailed description of the implementation)
- induces an implicit coupling between a class and all its super-classes (direct and indirect) as the modification of a super-class changes all its sub-classes.

The white box nature of inheritance goes against the black box nature of components. It seems that it would be better [Stein, 1987] to use composition which is a black box reuse mechanism to

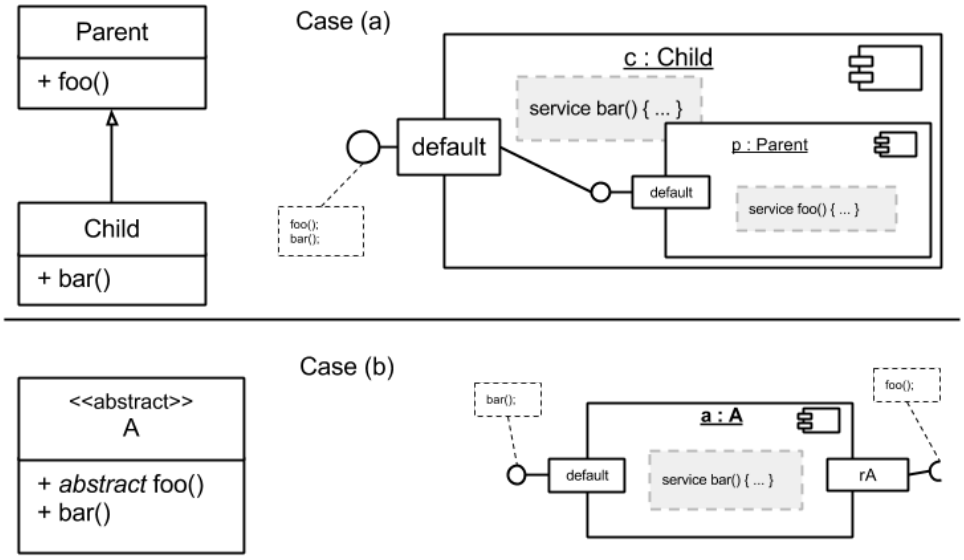


Figure 4.2 : Analogies between inheritance and composition

achieve incremental description and specialization. Such a solution was, for example, proposed by the authors of ComponentJ [Seco et Caires, 2000]. Figure 4.2 illustrates the composition solution for the following two cases:

Case (a) Building a subclass by adding new properties can be achieved by creating a composite that exports all the features of its internal components (*see* Fgure 4.2 case (a)).

Case (b) A class with abstract methods can be likened to a component with the required services (*see* Figure 4.2 case (b)).

A Java example in Listing 4.1 illustrates a class `Parent` which defines two methods named `bar` and `foo`. In the body of the `foo` method the `bar` method is invoked. Class `Child` inherits class `Parent` and it specializes its method `bar`. When the `foo` method is invoked on an instance of `Child`, the `Parents`'s code of `foo` is executed. Because the pseudo-variable `this` represents the current receiver, *i.e.* the instance of `Child`, method `bar` of class `Child` is executed.

However the same inheritance scenario realized with composition causes the "*initial receiver lost*" problem [Lieberman, 1986b], as it is shown in Listing 4.2[2]. In this example class `Child` emulates inheritance by the attribute `p` of type `Parent` to which it assigns a reference to an instance of `Parent`. Then `Child` defines a method `foo` which invokes `foo` on `p`, *i.e.* the `Parent`'s implementation of `foo`. In fact, the `foo` method of class `Child` implements a delegation of invocations similar to delegation connection defined in Section 3.2.4. When the `foo` method is invoked on an instance of `Child`, the `foo`'s

[2]In fact, the example uses aggregation (the private attribute `p` of class `Child`) which is a kind of internal composition [OMG, 2011b]

```
class Parent {
  public String foo() { return this.bar(); }
  public String bar() { return "bar() of Parent"; }
}

class Child extends Parent {     // inheritance
  public String bar() { return "bar() of Child"; }
}

public class Example{
  public static void main(String[] args){
    Parent p = new Child();
    System.out.println(p.foo()); // prints "bar() of Child"
  }
}
```

LISTING 4.1 : Executing an inherited code in a new context (the receiver environment), a Java example.

code is executed and invokes method `foo` on the `p` instance of class `Parent`. Because the pseudo-variable `this` represents the current receiver, *i.e.* the instance of `Parent`, method `bar` defined in class `Parent` is executed instead of the expected method `bar` of `Child`.

```
class Parent {
  public String foo() { return this.bar(); }
  public String bar() { return "bar() of Parent"; }
}

class Child {
  private Parent p = new Parent();        // composition
  public String foo() { return p.foo(); } // message forwarding
  public String bar() { return "bar() of Child"; }
}

public class Example{
  public static void main(String[] args){
    Child c = new Child();
    System.out.println(c.foo()); // ERROR: prints "bar() of Parent"
                                 //        instead of "bar() of Child"
  }
}
```

LISTING 4.2 : Composition and message forwarding to avoid inheritance leads to the "*initial receiver lost*" problem.

The same problem exists in the component context, where the child is a composite and the parent is its internal component, as we show in the top of Figure 4.3. ComponentJ proposes a solution for this problem in the component world. The solution (*see* the bottom of Figure 4.3) is based on additional required ports, delegations, a wrapping component and a feedback connection. Even if the ComponentJ's solution produces correct results its practical applicability is clumsy and requires a lot

of additional work.

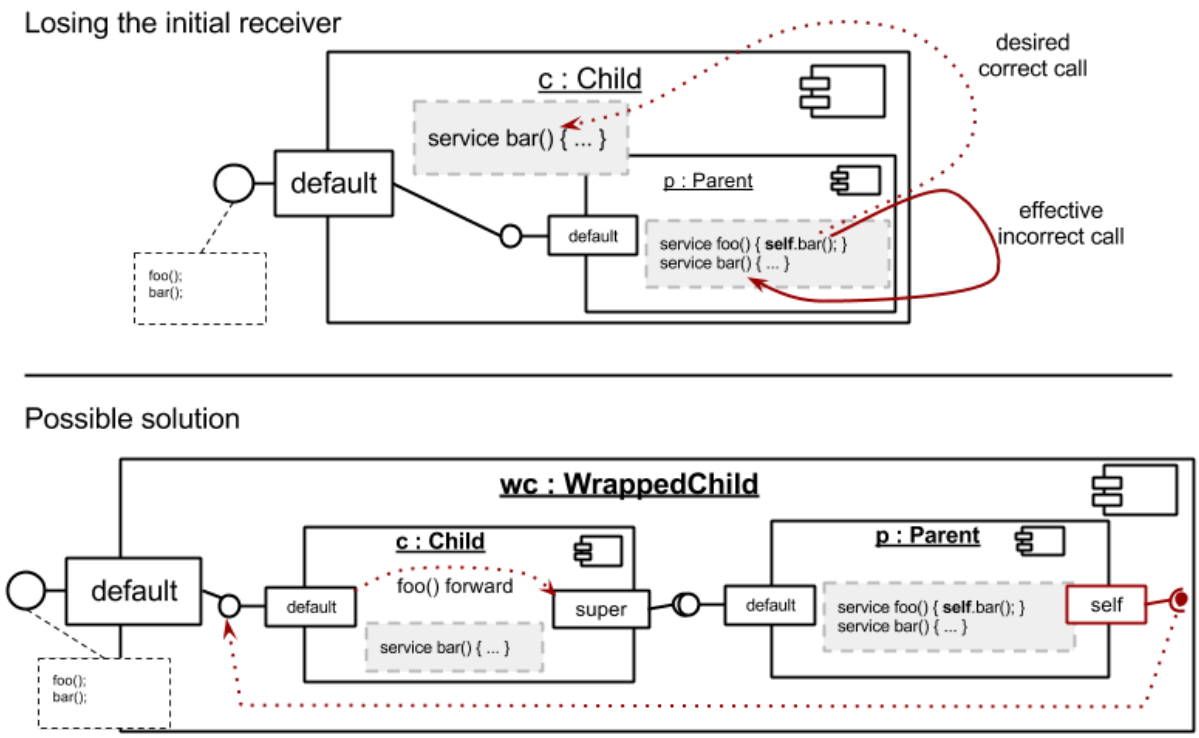


Figure 4.3 : An example of the initial receiver lose in case of composition and it possible solution as proposed in ComponentJ

It is the additional work needed (delegations[3] and interface repetition) when emulating inheritance with composition that makes its use difficult in practice. This stems from the fact that a composite is not necessarily a sub-type of types of all its internal components. In the component world, where descriptors define external contracts, the additional work contains redefinition of contracts, *i.e.* the descriptor of a child component has to define the same contract as the descriptor of a parent component. For example in Figure 4.3, a component c (an instance of a descriptor CHILD) has to provide the same as it is defined by the descriptor PARENT. In opposite to inheritance, the external contract of descriptors is not reused in case of composition.

This problem of composition can be generalized for the all cases when a structural definition, *i.e.* ports declarations or architecture, should be reused. We can say that structural reuse cannot be achieved with composition. Figure 4.4 shows an example of architecture reuse. The original architecture defined in the HTTPSERVER were reused and specialized by QUEUEDHTTPSERVER which adds a buffer component in between front-end and back-end internal components.

The above shows that inheritance is useful in case of descriptors, where it is advantageous and desired to be able to reuse definitions they contain. Our opinion is that composition and inheri-

[3] or redirecting, or forwarding

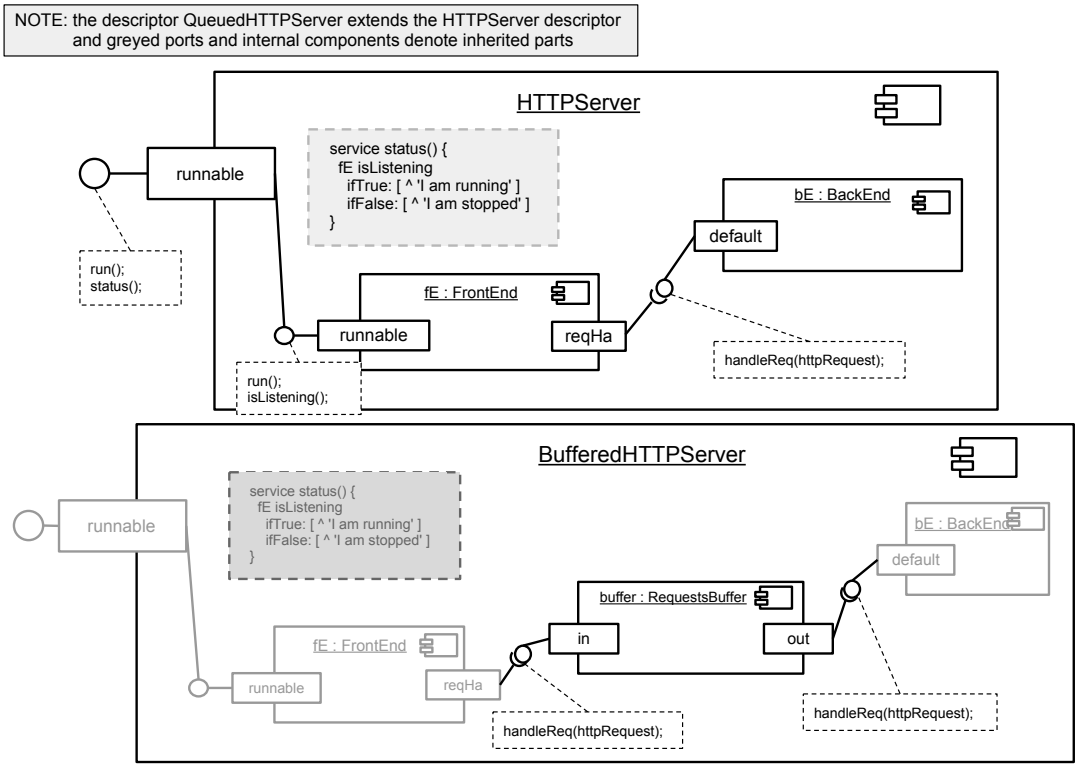


Figure 4.4 : An example of architecture reuse.

tance are complementary and that their combination is significantly more efficient especially when structural reuse is to be considered. We have thus designed an inheritance mechanism for COMPO in conjunction with composition to maximize software reuse capabilities and the language expressive power.

Moreover, in the component-based development context, as pointed by [Gamma *et al.*, 1995b], a set of available (off-the-shelf) components cannot cover all possible scenarios of usage, and therefore an adaptation mechanism is needed. Inheritance can be the mechanism, which enables programmers to easily extend or specialize such components.

As we have seen in Chapter 2, there are component-based approaches which somehow propose inheritance or inheritance-like [McVeigh *et al.*, 2006] mechanisms, but they have various limitations like: limitation to the architecture description side, limitation to the implementation side which is frequently not achieved with component-based languages or limitation to some part of components descriptions.

This chapter aims at contributing to that question by showing the interest of a specification and an operational integration of a full inheritance system, in the context of a component-oriented programming language (COL) that supports reuse of components' structure and behavior. By "*structure*", we mean descriptions of interfaces, ports and architectures; and by "*behavior*", we mean implementations of services that make them executable. Our language, COMPO, ranges in this category and

proposes components as run-time entities, instances of descriptors, as defined in Section 3.2.1. We introduce an inheritance link between descriptors on which we base an operational system to reuse, *i.e.* to **inherit**, **extend** and **specialize** descriptions.

In the rest of this chapter we address the specific questions of stating what, among port declarations, architectures and services definitions, can be inherited, extended or specialized, and how. We notably discuss the interest of enabling requirements extension or specialization, because required ports make dependencies explicit, reducing coupling between components and promoting understanding of components in isolation. We will consider various solutions, but will develop an answer to that question which goes in the direction of enabling covariant specialization [Boyland et Castagna, 1996], because it corresponds to the way human naturally think about concept classification [Ducournau, 2002] and it promotes modeling power.

4.2 Inheritance for structural and behavioral reuse

This section presents the rationale and the operational description of our descriptor-based inheritance embedded in COMPO. The inheritance mechanism of COMPO enables users to achieve *structural and behavioral reuse* that is:

1. to reuse the structure definition (external & internal contracts and architectures) captured in descriptors. In the rest we call this case *structural inheritance*.
2. to reuse the behavior definition (services implementations) captured in descriptors. In the rest we call this case *behavioral inheritance*.

In OOP, there is a structural (attributes and method declarations) inheritance and behavioral one (method bodies). All these declarations are subjects of the structure and behavior inheritance.

In a COMPO descriptor the behavior is given as a set of services definitions and the structure is given by port descriptions and by description of the architecture, *i.e.* the set of connection descriptions. We see no reason why these should not be subjects for the inheritance mechanism in COMPO. Therefore we chose the following:

Choice 23 *Port descriptions, architectures (connection descriptions sets) and services are subjects of* COMPO *inheritance*

As in OOP, where a sub-class may extend and specialize its super-class. We desire to be able to extend and specialize component descriptors. The “extends” operation means adding new subjects to the parent’s definition. The “specialize” operation means the modification of subjects previously defined without introducing new subjects. These operations can be separated into the two following categories:

1. The operations affecting the external contract of a descriptor, *i.e.* addition and specialization of ports.

2. The operations affecting the architecture of a descriptor, *i.e.* addition and specialization of internal components and connections.

In case of architecture, the extension and specialization are sometimes related. For example adding new internal component usually requires some re-connections, *i.e.* specialization of original connections.

The incremental (extend) description philosophy of inheritance does not provide means for removing subjects, although an inheritance-like mechanism called *resemblance* [McVeigh *et al.*, 2006] has been proposed in the component context. We believe that the philosophy of removing does not match well with the way people usually think of and design hierarchies of concepts.

4.2.1 Multiple inheritance, yes or no?

We argue that integrating multiple inheritance into COMPO is undesirable for the following reasons:

1. In the world of objects, different languages actually have different expectations for how multiple inheritance (MI) works. For example, how conflicts are resolved and whether duplicate bases are merged or redundant. Before we can even think about MI for a COL, we have to do a survey of all the OOP languages, figure out the common concepts, and decide how to express them in a language-neutral manner. Then we would have to transfer these concepts from the object world into the world of components, where even more conflicts may occur, because the contracts of components are richer than contracts of objects.

2. The number of places where MI is truly appropriate is actually quite small. In many cases, it is possible to use encapsulation and delegation. Moreover, the alternatives for MI like mixins [Bracha et Cook, 1990] or traits [Curry *et al.*, 1982] should also be considered.

3. Multiple inheritance implementation injects a lot of complexity into the implementation. This complexity impacts dispatch, port access, identity comparisons, reflection and probably lots of other places.

The amount of work needed to integrated multiple inheritance into a COL seems to be inadequate compared to the value added, thus, we chose the following.

Choice 24 *The inheritance mechanism in* COMPO *is single inheritance.*

In fact, this choice is very usual in the world of components. We are aware of few component-based approaches integrating multiple inheritance. For example, SOFA model has chosen concatenation inheritance mechanism for its *frames* [Oplustil, 2002] to solve name conflicts related to multiple inheritance. UML also allow multiple inheritance, but it provides no explicit solution or recommendations for well known issues and ambiguities, such as the diamond problem [Boyland et Castagna, 1996].

4.3 Descriptors and basic inheritance

In this section we introduce the inheritance link between descriptors. In parallel to OOP, where the inheritance link between classes organizes them into a hierarchy of sub-classes and super-classes, we define sub-descriptors and super-descriptors. In this chapter we also use terms *child* and *parent* which are nothing more than aliases for terms sub-descriptor and super-descriptor respectively.

Definition 16 (Sub-descriptor) *A descriptor may extend and specialize another descriptor, such a descriptor is then called a* sub-descriptor.

Definition 17 (Super-descriptor) *If a descriptor* C *is defined as a sub-descriptor of a descriptor* D*, then we say that* D *is a* super-descriptor *of* C.

Having the terminology set and in accordance with the arguments we have presented in Section 4.2 we make the following choice:

Choice 25 *A sub-descriptor inherits all subjects of its super-descriptor (its parent), i.e. all ports descriptions, the architecture and all services definitions.*

The natural consequence of the Choice 25 is that we do have common problems of inheritance, such as the fragile base class problem (*see* Section 4.1). The intent of this chapter is not to speak about these problems, but discuss the new issues, which arise when using inheritance as the reuse mechanism for a COL.

In COMPO, a new sub-descriptor is defined by the **`extends`** operator. For example, to create a descriptor of an extended calculator component, we define a new descriptor EXTCALC as a sub-descriptor of an existing descriptor CALC by the following declaration: `Descriptor ExtCalc extends Calc {...}`.

For code factorization and reuse purposes we enable to create abstract descriptors

Definition 18 (Abstract descriptor) *When it is declared that an instance of a descriptor offers a service, but the service is not defined, then the descriptor is considered as an abstract component descriptor.*

Although, an abstract descriptor cannot be instantiated its sub-descriptor may provide implementation of the missing service and thus make itself instantiable, *i.e.* non-abstract.

4.3.1 The ExtCalc Example

In the previous chapter in Section 3.3.4, we presented the substitution mechanism of COMPO on the example of the calculator component which is substituted by the extended calculator component. The components were instances of descriptors CALC (*see* Listing 3.4) and EXTCALC (*see* Listing 3.8) respectively. Both components are shown in Figure 4.5.

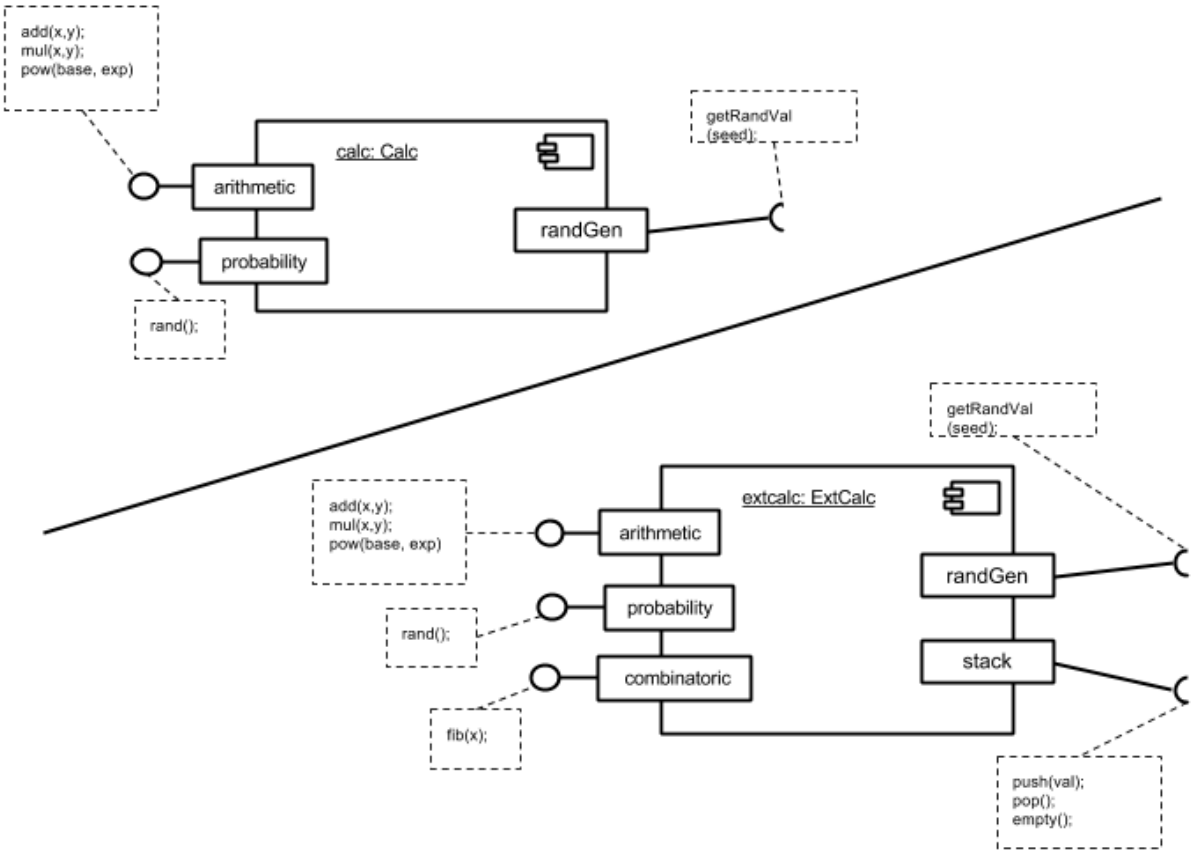


Figure 4.5 : The instances of the CALC (defined in Listing 3.4) and the EXTCALC (defined without inheritance in Listing 3.8 and with inheritance in Listing 4.3.)

The descriptor CALC defines a very simple `calculator` component providing three arithmetic services and one service for generating random numbers. `calculator` provides these services through ports `arithmetic` and `probability`. To fulfill its commitment to provide the probability service `rand`, the component requires a random generator via required port `randGen`. The definitions of services `add` and `mul` show the use of arithmetic operators and the returned value. Service `pow` shows a program flow control structure **`for`**, an assignment and a service invocation. An example of a required service invocation is captured in the definition of the `rand` service. Listing 3.5 and Figure 3.6 show an example of how the `calculator` component can be instantiated and used.

The EXTCALC descriptor defines the same as descriptor CALC and through a new port `combinatoric` it provides a new service `fib` to calculate Fibonacci numbers. In order to calculate the Fibonacci numbers, services `push`, `pop` and `empty` of a stack component are required through port `stack`.

It is obvious that EXTCALC descriptor should be defined as a sub-descriptor of descriptor CALC. In Listing 4.3 we present descriptor EXTCALC2 which uses the inheritance mechanism of COMPO to

extend descriptor CALC with declaration of ports `combinatoric` and `stack` and with the definition of service `fib`.

```
Descriptor ExtCalc extends Calc {
  provides {
    combinatoric : { fib(x); }
  }
  requires {
    stack : { push(val); pop(); empty(); }
  }
  service fib(x) {
    | tot a |
    tot := 0 ;
    while(stack.empty() == false) {
      a := stack.pop()
      if(a < 1) { tot := tot + 1; }
      else {
         stack.push(a - 1);
         stack.push(a - 2);
      }
    }
    return tot;
  }
}
```

LISTING 4.3 : The EXTCALC2 descriptor is defined as a sub-descriptor of descriptor CALC (defined in Listing 3.4).

4.4 Addition & specialization of services

The services are units of behavior which a component provides. For example, the `calculator` component provides services for addition, subtraction, multiplication and division. The services are subjects of behavioral inheritance similarly to methods being subjects of behavioral inheritance in OOP.

Behavioral inheritance means the ability to access and modify the implementation of the parent. To be able to inherit, extend and specialize the behavior defined by a component descriptor we make the following choice:

Choice 26 *A sub-descriptor can introduce new services and its instances can invoke, redefine and specialize services defined by its super-descriptor.*

This gives us the ability to define behavior the that is specific to a particular sub-descriptor, *i.e.* achieve polymorphism of descriptors. A new service can be introduced in a sub-descriptor as it is illustrated in Listing 4.4 where the descriptor B extends descriptor A with a new service `bar`.

An inherited service can be redefined or specialized in a sub-descriptor. The difference between redefinition and specialization is that in case of specialization, the super-descriptor's implementation

```
Descriptor A {
   provides { default : { foo(); } }
   service foo() { return 0.0; };
}
Descriptor B extends A {
   service foo() { super.foo(); return 0; } /* specialization */
   service bar() { ... }                    /* addition */
}
```

LISTING 4.4 : Specialization and addition of services.

of the service being specialized is invoked (manually by user) in the code of the new implementation of the service.

Services specialization raises a new question: *How to invoke services of super-descriptors?* An inspiration for answering this question can be found in Java's inheritance mechanism. Suppose that we have defined two Java classes `A` and `B`, where class `B` is a sub-class of class `A` and `B` specialize an inherited method `foo`. In the code of `B`'s `foo` method we need to invoke method `foo` of `A`. To do so, Java uses the `super` pseudo-variable of Java. In Java, the pseudo-variable `super` is statically computed and represents a super-class of the class in which the method using `super` is defined. Back to the ArchJava example, we can use `super.foo()` in the code of `B`'s `foo` to invoke `foo` of `A`.

In the previous chapter, we have made the Choice 14 saying that all service invocations are always made via ports. Consequently, the invocations of services of a super-descriptor should be made through a port. The following choice captures this consequence:

Choice 27 *Every sub-descriptor has, by default, the* `super` *internal provided port. Service invocations sent through this port are looked up starting from the super-descriptor of the descriptor owning the service in which code the service invocation is emitted.*

For example, the descriptor B (in Listing 4.4) extends descriptor A and specializes A's service `foo` by use of the `super` port. The Choice 27 mentions the service lookup mechanism which handles service invocations of services which are not defined by a receiver component. We detail this mechanism in the following sub-section.

4.4.1 The service lookup mechanism

Service invocations in COMPO resembles message sending in OOP. Objects in OOP communicate by sending messages. But, what exactly happens when an object receives a message? First, there is not a universal answer to that question, because there is not a universal object-oriented language. Therefore, to answer this question we have take a look on a language which is considered to be a pure object-oriented language, such as SMALLTALK [Ingalls, 1981].

> "*When a* SMALLTALK *object receives a message, the class of the receiver looks up the method to use to handle the message. If this class does not implement the method, it asks its super-class, and so on, up the inheritance chain, as shown in Figure 4.6. When the method*

is found, the arguments are bound to the parameters of the method, and the virtual machine executes it.

It is as simple as that, but there is a question that needs some care to answer: What happens if the method we are looking for is not found? *Suppose we send the message* `foo` *to our ellipse. First the normal method lookup would go through the inheritance chain all the way up to class* `Object` *looking for this method. When this method is not found, the virtual machine will cause the object to send itself the* `doesNotUnderstand: #foo` *message. So the lookup starts again from the class* `EllipseMorph`*, but this time searching for the method* `doesNotUnderstand:`*. As it turns out, class* `Object` *implements* `doesNotUnderstand:`[4]*. This convoluted path offers developers an easy way to intercept such errors and take alternative action. One could easily override the method* `doesNotUnderstand:` *in any sub-class of* `Object` *and provide a different way of handling the error*[5]*.*" [Black *et al.*, 2009].

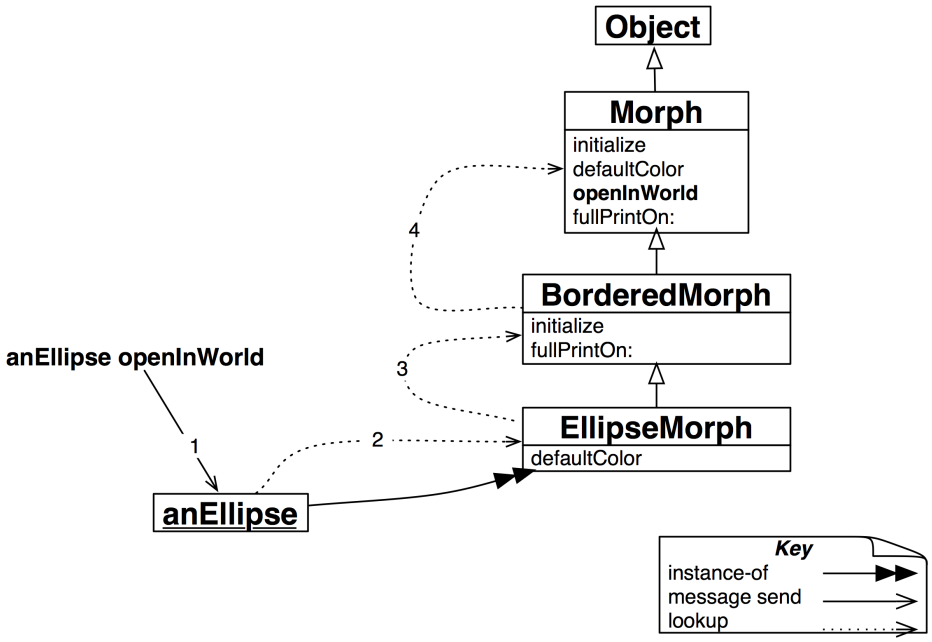


Figure 4.6 : An example of the method lookup mechanism in SMALLTALK. The mechanism follows the inheritance hierarchy.

Inspired by the method lookup mechanism of SMALLTALK we define the service lookup mechanism of COMPO as follows:

[4]This method will create a new MessageNotUnderstood object which is capable of starting a Debugger in the current execution context.

[5]In fact, this can be an easy way to implement automatic delegation of messages from one object to another. A Delegator object could simply delegate all messages it does not understand to another object whose responsibility it is to handle them, or raise an error itself!

Definition 19 (The service lookup mechanism) *When a provided port receives a service invocation, it treats the invocation according to the Algorithm 3. In the case of lookup, the port looks up the requested service in the descriptor of the component that owns the port. If this descriptor does not implement the service, the port asks its super-descriptor, and so on, up the inheritance chain. When the services is found, the arguments are connected to the parameters ports of the service, and the service is executed. When the service is not found, then the* `doesNotUnderstand` *service of the port is executed.*

Algorithm 4 is a modification of the Algorithm 3 (*see* Section 3.3.2 of the previous Chapter) taking into account the service lookup mechanism as it is defined in Definition 19. As we will see in the next chapter, ports are also components having their own services and the `doesNotUnderstand` service is one of them. It actually enables the users to customize the method lookup, because it is possible to design new kinds of ports. In fact, the possibility to customize the method lookup was inspired by the method lookup objects technique [Vraný *et al.*, 2012].

Algorithm 4: Taking into account the service lookup mechanisms when receiving a service invocation via a provided port

```
Data:
    i : a service invocation
    p1 : the provided port through which i was received
if selector(i)∈ interface(p1) then
    if p1 is delegated to a provided port p2 then                    /* case 1 */
        delegate i through p2;
    else
        if descriptor(component(p1)) possesses the service demanded in i then   /* case 2 */
            execute s
        else
            if descriptor(component(p1)) does not implement
            the service demanded in i then                           /* case 3 */
                d := descriptor(component(p1));
                while super-descriptor(d) does not implement the service demanded in i do
                    d := super-descriptor(d);
                if d != null then
                    execute s of d;
                else                                   /* does not understand error */
                    execute does−not−understand of p1;
            else                                                 /* p1 not connected */
                error case;
else                                   /* service is not listed in the interface */
    error case;
```

4.5 Addition & specialization of provided port descriptions

The ability to add a new port declaration into a sub-descriptor improves modeling power of the language. For example, sub-descriptors are able to export an internal behavior via newly added ports. Such an export does not break the encapsulation of the internal component, because it exports behavior which has already been made public (*i.e.* provided via external port). We show an example of such an export in Listing 4.5. The descriptor CONTROLABLEFRONTEND extends a FRONTEND descriptor with a new description of port `control` and a delegation connection which delegates service invocation sent via the port to the `control` port of the internal component `regRecv` (instance of descriptor REQUESTRECEIVER.) The example is shown in Figure 4.7. With the Choice 28, we allow for addition of new provided ports in sub-descriptors.

Choice 28 *A sub-descriptor can introduce a new provided port description.*

```
Descriptor ControlableFrontEnd extends FrontEnd
{
   provides {
      control : {start(); isRunning(); stop()}
   }
   architecture {
      delegate control@self to control@regRecv;
   }
}
```

LISTING 4.5 : The CONTROLABLEFRONTEND descriptor. Extends a FRONTEND descriptor with a new provided port named `control`. Instances of the both descriptors are shown in Figure 4.7

Similarly to OOP, where the name of a new attribute cannot clash with the names of super-class attributes, the majority of component-based approaches with inheritance support does not allow for port-names clashing. In accordance with that, we chose the following:

Choice 29 *A name of a newly added port of a sub-descriptor cannot clash with existing port names (inclusive inherited port names).*

If a sub-descriptor introduces a port description with a clashing name, then it means that the sub-descriptor specializes the inherited port description with that name. In the previous chapter in Section 3.2.2, we have defined that ports are described by the *names*, a *visibility*, a *role* and an *interfaces*. The question of port specializations has then the three following sub-questions:

1. Does it make sense to specialize the role?
2. Does it make sense to specialize the visibility?
3. How do we specialize interfaces?

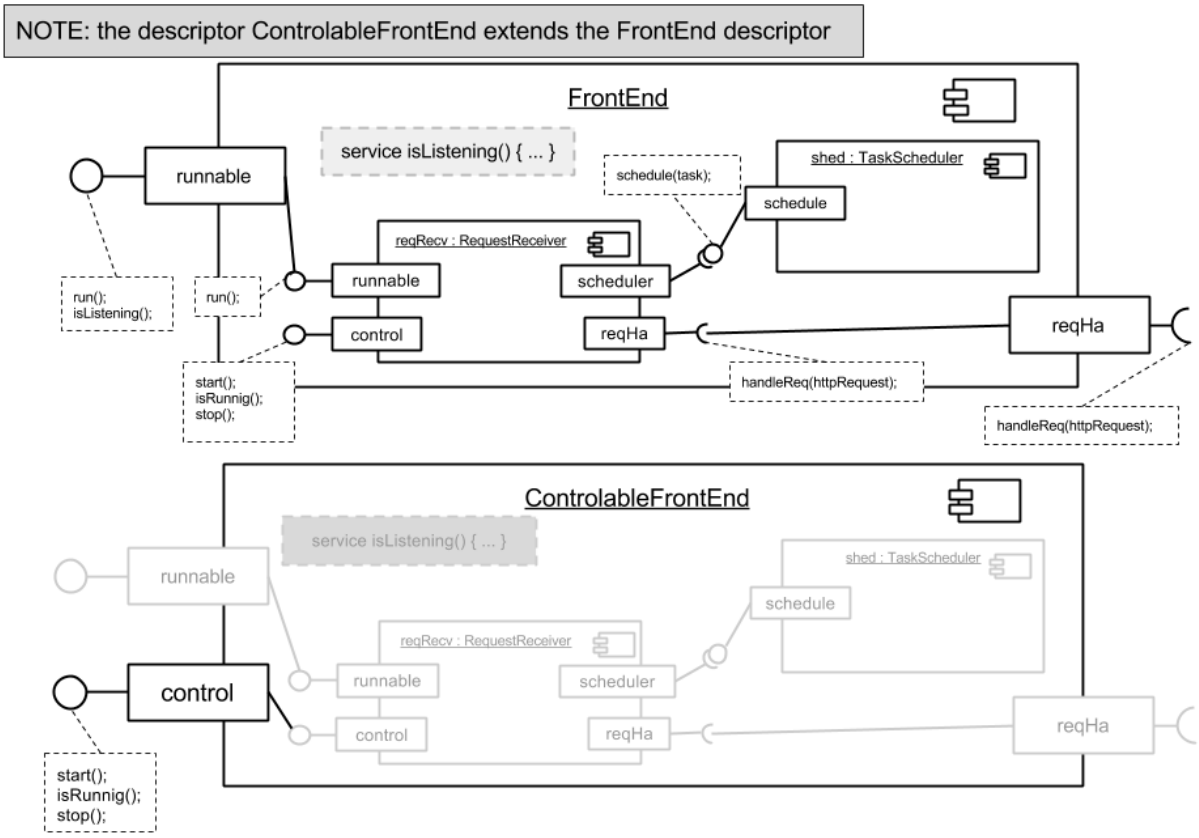


Figure 4.7 : Descriptor CONTOLABLEFRONTEND (*cf.* Listing 4.5) exports the controlling behavior of the inherited internal component `reqRecv` via the newly added port `control` and a delegation connection. Greyed parts denote inherited subjects.

The specialization of port roles comes into consideration in two directions. A change from the provided role to the required role does not make sense. It could make sense in the opposite direction, *i.e.* from the required role to the provided role. Indeed, this kind of role specialization means that a sub-descriptor does no longer require the services exported via this port and therefore it is equivalent to removing that port, which is in contradiction to the incremental (extend) description philosophy of inheritance. Thus, there is no reason for allowing specialization of the roles of ports.

The specialization of the visibility contains two cases: (i) change from the external to the internal visibility; and (ii) change from the internal to the external visibility. In fact, the first case is comparable to removing a port, because it removes the description of the port from the external contract definition of a sub-descriptor, which is also in contradiction to the incremental (extend) description philosophy of inheritance. The second case affects the internal contract of a sub-descriptor. The architecture of the super-descriptor is designed on the basis of the internal contract and a change to the contract may cause internal incompatibilities. Therefore, we forbid the visibility specialization and we make the following choice.

Choice 30 *It is not possible to specialize the visibility nor the role of an inherited port description in a sub-descriptor.*

In opposite to the role and visibility specialization, the specialization of interfaces is desired and needed, because it makes possible to publish new services via existing (inherited) ports. Therefore, we make the following choice:

Choice 31 *A sub-descriptor can specialize the list of service signatures (the interface) of an inherited port description.*

We remind (*cf.* Section 3.2.2) that the interface of a port is a set of service signatures which could be given in three forms:

- as an explicit list (we call such a list an *anonymous interface*), for example the `default` port declaration in Listing 3.1
- as a named interface, e.g. `printer : IPrinting` where the interface `IPrinting` was created with the statement: `interface IPrinting {print(text); ....};`
- as a descriptor name; in this case, the list is the list of signatures of services associated to default provided port of the descriptor. For example, the `fE` port declaration in Listing 3.1.

There are three scenarios how to specialize the list of service signatures of an inherited port:

1. a specialization by adding new service signatures to its list of service signatures (*i.e.* extending an anonymous inherited interface). The RESTARTABLEFRONTEND descriptor in the example in Listing 4.6 shows the specialization of the `control` port defined by the CONTROLABLEFRONTEND super-descriptor. The specialization is used in order to provide the `restart` service defined by the descriptor RESTARTABLEFRONTEND.
2. a specialization using a named interface. In this case the set of service signatures defined in the named interface has to be a super-set of the set of service signatures used to describe the original port. A specialization of a port named `portA` looks like: `provides { portA : Ispec2}`. In the super-descriptor, `portA` could have been declared by the two following statements:

   ```
   provides { portA : { ser1(); ser2() }} /* anonymous interface */
   ```

   or

   ```
   provides { portA : Ispec1 } /* named interface */
   ```

   The `Ispec1` interface was defined with the statement `interface Ispec1 { ser1(); ser2(); }` and the `Ispec2` interface extends `Ispec1` with a service signature: `interface Ispec2 extends Ispec1 { ser3() }`. The named interface `Ispec2` defines the set of service signatures, which is a super-set of a set representing the anonymous interface or the set given by the `Ispec1` interface of the original `portA` port. Therefore it can be used for the specialization.

3. a specialization using an anonymous interface. This case is very similar to the previous one, *i.e.* the set of service signatures defined in an anonymous interface has to be a super-set of the set of service signatures used to describe the original port.

```
Descriptor RestartableFrontEnd extends ControlableFrontEnd
{
   provides {
      control : { restart(); }
   }
   service restart() {
      reqRecv.stop();
      reqRecv.start();
   }
}
```

LISTING 4.6 : The RESTARTABLEFRONTEND descriptor. Specializes the `control` port of CONTROLABLE-FRONTEND descriptor (*cf.* Listing 4.5).

The addition and specialization of ports descriptions changes the external contract of a super-descriptor. From the substitutability perspective, any changes made on the external contract are risky. Luckily changes on the provided part of the external contract are not critical, simply because of the nature of provisions. Changes on the required part of the external contract modify dependencies of descriptors. Therefore extension and specialization of required ports is considered as a risky operation, as we will detail in the next section.

4.6 Addition & specialization of external required ports descriptions

In general the rules about addition and specialization of external required ports descriptions are similar to those we present in the previous section about provided ports. But, in opposite to provided ports, the question whenever we allow or not for addition and specialization of external required ports descriptions needs some care to answer.

The usual practice with differential descriptions is incrementation. A colored circle is like a circle and, in addition, it has a color. The reverse perspective saying that a circle is a colored circle without a color does not conform to the way people usually think. To have a color assigned is a requirement for a colored circle. To make requirements explicit is one characteristic of the component-based approach. In the object-oriented world, it is common to define an additional attribute in a sub-class. It is however an issue to know whether it is allowed to define a new required port on a sub-descriptor. In fact it raises the same issue in both worlds: *it possibly breaks child-parent substitutability* [Spacek *et al.*, 2012]. More precisely, it may lead to incorrect substitutions.

In OOP, substitutability between instances of classes and instances of their sub-classes is guaranteed. To be more precise, interface compatibility is guaranteed, behavior compatibility is still not guaranteed, as pointed by [Martin, 2002]. However the situation is different in case of components, where requirements are explicit.

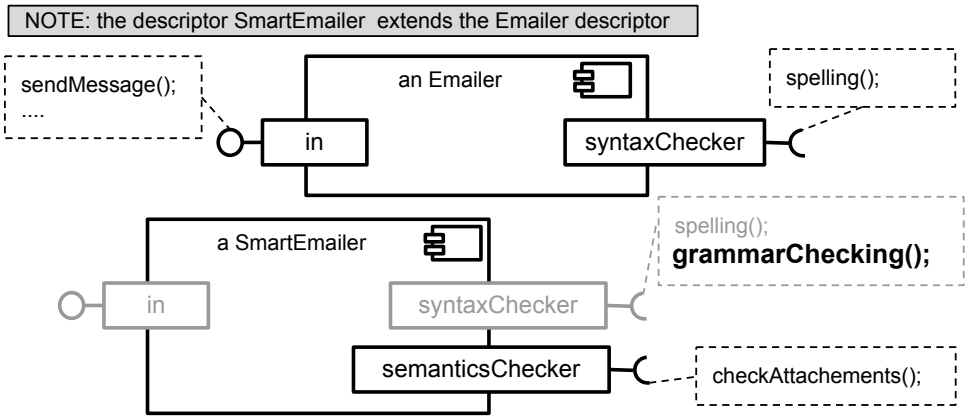


Figure 4.8 : An example of an extension and specialization of required ports. Grayed parts of the figure illustrate inherited parts.

In CompJava, a component type inherits all ports of its super-type and it may extend the interfaces of inherited provided ports or may add provided ports. Addition of required ports is not allowed due to the substitutability policy of the CompJava model (*cf.* Section 3.3.4).

Port specialization achieved using interface redefinition is implemented in SOFA model by the following statement

```
frame ComponentName inherits InheritedCompName changes
 InterfaceIntance1:: OriginalInterfaceType1 => NewInterfaceType1
```

SOFA constraints the above statement by saying that the interface type `NewInterfaceType1` is a sub-type of the `OriginalInterfaceType1` interface type.

The problem of addition and specialization of external required ports has in fact three possible solutions:

1. Forbid extension of sub-descriptor with additional requirements. But, requirements have been made explicit in components and they are considered as important entities to make it possible to introduce new connections. Therefore it is undesirable to limit expressive power of modeling by forbidding extension and specialization of requirements. For example, without possibility to add a required port it is complicated to design the `Emailer` example shown in Figure 4.8.

2. Constrain substitutions - define a rule saying that an original component can be substituted by a new one, only if the new one provides at least the same and requires at most the same as the original one, *i.e.* to follow the Liskov substitution principle [Liskov et Zilles, 1974].

3. Allow additional requirements and delegate responsibility for additional requirements satisfaction to the language users, while providing verification support for substitutions.

With COMPO all alternatives could have been implemented, but since the language is oriented toward modeling flexibility, we have experimented with the third alternative. We will thus support covariant specialization if and when needed, because it corresponds to the way human naturally think differential description [Ducournau, 2002]. For the sake of expressive power, we choose to enable declaration of new required ports in sub-descriptors. For the same reason we enable the covariant specialization of required ports in sub-descriptors.

Choice 32 *A sub-descriptor can introduce a new external required port description or it may specialize the interface of an inherited description of an external required port.*

The Choice 32 prevent static checking of the correctness of substitutions and of course a different choice could be made. But, without such capabilities, the modeling power of the language would be limited. For example, we would not be able to extend the EMAILER descriptor shown in Figure 4.8 with a new required port `semanticsChecker` or specialize its required port `syntaxChecker` with a new required service signature `grammarChecking()`. In order to combine additional requirements and substitutability we propose an alternative approach with a dynamic checking of substitution correctness.

Substitutions in COMPO are supported by the replace routine which were detailed in Section 3.3.4. Here we remind the substitutability definition (*see* Definition 14). A component `a` is substitutable with a component `b` if the descriptor of `a` is compatible with the descriptor of `a` and all requirements of component `b` will be satisfied after the substitution.

Our inheritance mechanism does not apply any restrictions to implicitly guarantee substitutability. However, we are providing a support service `newCompatible` to return a component compatible with the super-descriptor. The service is automatically created for each sub-descriptor which extends its parent with additional requirements. The service has a unique parameter, an array of pairs *port-component* and it is able to create an instance, which is substitutable with instances of the super-descriptor. That is, all additional requirements are satisfied by connections to components given in the array argument. The service shows a little bit from the reflection features of COMPO. As we will see in the next chapter, descriptors are also components having their own services and the `newCompatible` service is one of them. The following example shows an application of the replace routine and the `newCompatible` service.

4.6.1 The DynamicHTTPServer example

The dynamic http server example presents an application of the replace routine (*cf.* Section 3.3.4) and the `newCompatible` service presented above. In this example, we create a sub-descriptor of the QUEUEDHTTPSERVER descriptor (*see* Listing 4.8) called DYNAMICHTTPSERVER. The DYNAMICHTTPSERVER descriptor (*see* Listing 4.7) extends the QUEUEDHTTPSERVER descriptor with two new services `switchToStandardQueue` and `switchToRandomQueue`, in order to be able to dynamically substitute the `queue` internal component in server instances. The `switchToRandomQueue` service causes that the instance of the REQUESTQUEUE descriptor (connected to the internal required port `queue` of the server) is substituted with an instance of the RANDOMREQUESTSQUEUE descriptor. The `switchToStandardQueue` switches the queues back. The substitution is illustrated in Figure 4.9.

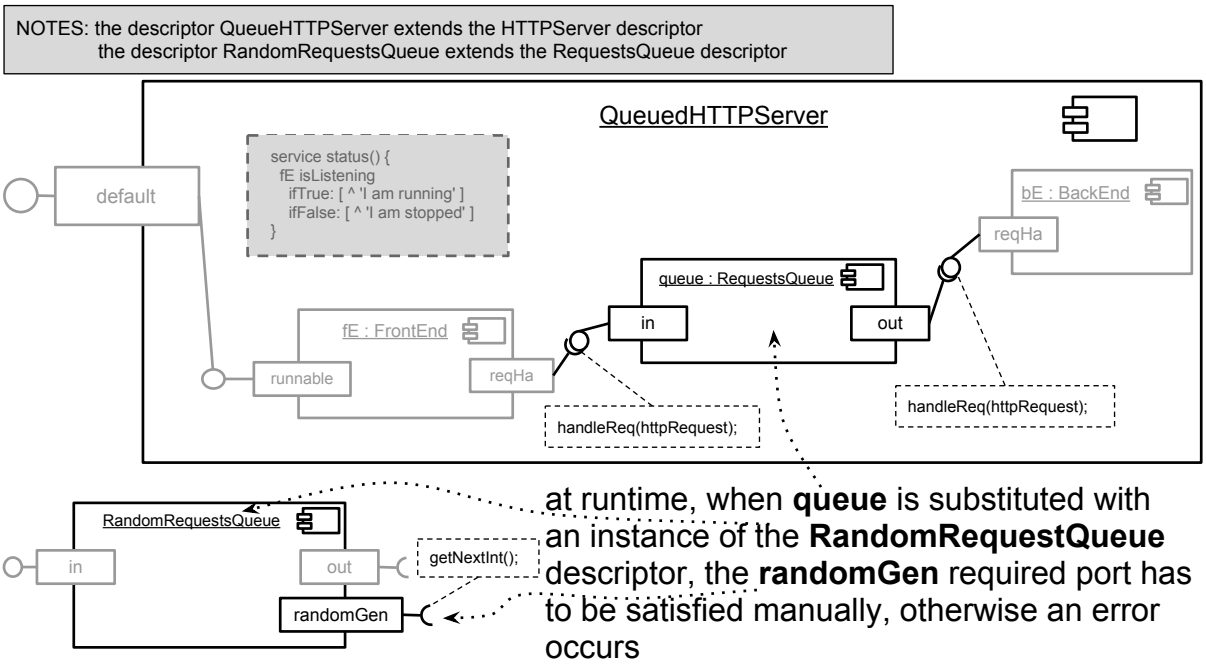


Figure 4.9 : Dynamic substitution with a sub-descriptor having additional required port may lead to unsatisfied requirement in the architecture. Grayed parts of the figure illustrate inherited parts.

It is important to note that the RANDOMREQUESTSQUEUE descriptor is a sub-descriptor of the REQUESTSQUEUE descriptor, which has an additional required port to connect a random generator component (instance of the RANDOMGENERATOR descriptor shown in Listing 4.7.) This additional requirement makes the substitution complicated because the original architecture of the QUEUED-HTTPSERVER descriptor was not designed for this kind of queue.

In order to make substitutions possible, the `switchToRandomQueue` service uses the `newCompatible` service of the RANDOMREQUESTSQUEUE descriptor to create an instance of RANDOMREQUESTSQUEUE compatible with instances of the REQUESTSQUEUE descriptor and thus easily substitutable in the architecture. The `newCompatible` takes as an argument an array of pairs *port-component*, in our case the array contains only one pair: `randomGen-RandomGenerator.new()`. The pairs in the array are used to satisfy the additional requirements of the RANDOMREQUESTSQUEUE descriptor. To perform the substitution the service uses the replace routine.

```
Descriptor RequestsQueue
{
   provides { in : { handleReq(httpRequest); }
   requires { out : { handleReq(httpRequest); }
   ...
}

Descriptor RandomRequestsQueue extends RequestsQueue
{
   requires { randomGen : { getNextInt(); }
   ...
}

Descriptor RandomGenerator {
   provides { generator : { getNextInt(); }
   ...
}

Descriptor DynamicHTTPServer extends QueuedHTTPServer
{
   service switchToRandomQueue()
   {
      | pair randomQueue |
      pair := Pair.new('randomGen',(RandomGenerator.new()));
      randomQueue := RandomRequestsQueue.newCompatible(Array.newWith(pair));
      replace queue with randomQueue;
   }

   service switchToStandardQueue()
   {
      replace queue with RequestsQueue.new();
   }
}
```

LISTING 4.7 : An example of unsatisfied required port problem and its solution using the `replace` routine and `newCompatible` service. The DYNAMICHTTPSERVER descriptor can dynamically substitute the queue internal component in its instances. The RANDOMREQUESTSQUEUE descriptor extends the REQUESTQUEUE descriptor with an additional required port to which an instance of the RANDOMGENERATOR descriptor should be connected.

4.7 Extension & specialization of architectures

In case of the internal architecture description (architecture in short), the extension and specialization are usually related. For example, addition of a new internal component very often requires some re-connections, *i.e.* specialization of original connections, as we show in Figure 4.9. In the example, an instance QueuedHTTPServer descriptor is like an instance of HTTPServer descriptor, but instead of having a direct connection between the `fE` component (instance of FrontEnd descriptor) and the `bE` component (instance of BackEnd descriptor), an adaptor, here a queue, is inserted in between `fE` and `bE` and the original `bE-fE` connection is replaced by two connections to and from the queue.

In software evolution, architectures often need to be reused and the used description language should support such a feature [Cioch *et al.*, 2000]. This observation is based on general experience with building object-oriented applications, where it is common that a class describes objects, which are composed of many objects communicating together. Such classes are often sub-classed. The sub-classes modify and extend original composition and communication system of their parent. This also applies in the component world and we can think of many situations in which a new architecture will be based on an existing one.

Although useful, architectures reuse by extension and specialization of internal components and connections has to be used carefully. These operations carry certain risks because they may destroy an internal architecture defined by a super-descriptor. For example a missing connection or usage of an incompatible internal component may make the composite in-operational, as we have detailed in Section 3.3.4. The extension and specialization of architectures involves the following operations:

- addition of new internal components
- replacement of inherited internal components
- replacement of connection descriptions

To be able to achieve these operations, a sub-descriptor has to have access to the architecture specification of its super-descriptor. For that reason, we make the following choice:

Choice 33 *A sub-descriptor inherits the architecture of its super-descriptor. It may introduce new internal required ports descriptions and new connection descriptions. It may specialize the inherited connection descriptions and the interfaces of inherited internal required ports descriptions.*

In the previous chapter in Section 3.3.3 we made the Choice 21 saying that: a composite communicates with its internal components through internal required ports, one per an internal component. Therefore, the addition of new internal components is realized by addition of new internal required ports which follow the same rules which apply for external required ports (*see* the previous section.) Specialization of an inherited internal component in a sub-descriptor can be achieved by modifying the interface description of the internal required port associated with the internal component. The descriptor of `PriorityQueuedServer` in Listing 4.8 specializes an inherited internal component `queue` by describing it with the `PriorityRequestsQueue` descriptor.

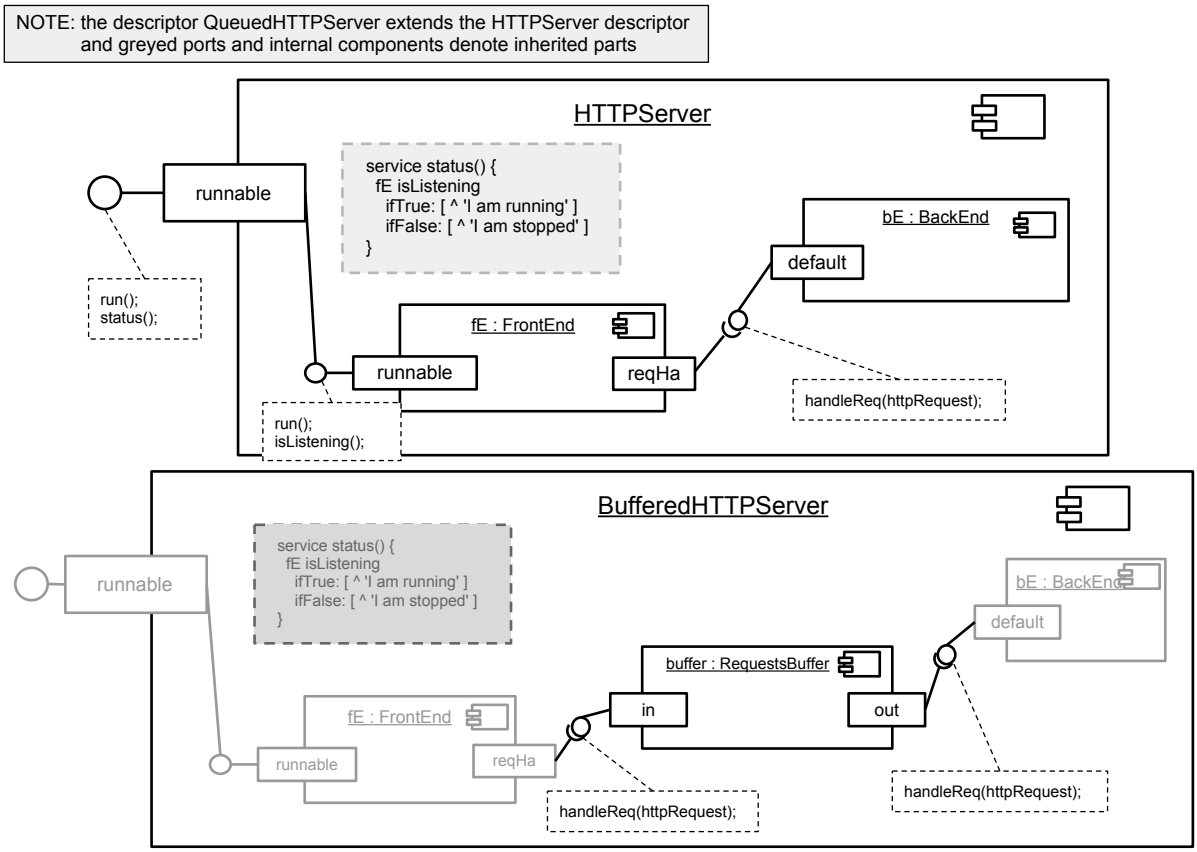


Figure 4.10 : Specialization and extension of an internal architecture. Grayed parts of this figure illustrate inherited parts.

Newly added connection description cannot clash with inherited connections, for this case a specialization of connection descriptions is used. When specializing connection descriptions, it is sometimes hard for the architects to determine if a new connection description replaces the inherited one or if it just defines a new connection. For that reason we use the combination of the **`connect-to`** and **`disconnect-from`** statements to specialize inherited internal connections. The disconnections clearly specify the connections to be replaced making the specialization well defined. The statements have the following syntax: `disconnect <port-address> from <port-address>` and `connect <port-address> to <port-address>` (disconnections and connections were explained in Section 3.2.4.) Every time the `disconnect` statement is used, the removed connection has to be superseded by a new connection. Extension and specialization of internals is illustrated in Figure 4.10 with COMPO code in Listing 4.8.

```
Descriptor QueuedHTTPServer extends HTTPServer
{
   internally requires {
      queue : RequestsQueue
   }
   architecture {
      connect queue@self to default@(RequestsQueue.new());
      disconnect backEnd@fE from default@bE;
      connect backEnd@fE to in@queue;
      connect out@queue to default@bE;
   }
}

Descriptor PriorityQueuedServer extends QueuedHTTPServer
{
   internally requires {
      queue : RandomRequestsQueue;
   }
   architecture {
      connect queue@self to default@(RequestsQueue.new());
      connect queue@self to default@(RandomRequestsQueue.new());
   }
}
```

LISTING 4.8 : Specialization and extension of an internal architecture.

4.8 Related work

The study made in Chapter 2 shows that if (at all) inheritance is present the it is interpreted differently by each component-based approach. For example, the meta-model of UML defines that class `Component` inherits from class `Class`, giving the possibility to participate in generalization relationships to class `Component`, *i.e.* to inherit all properties (attributes and methods) and all parts (internal components) defined in another class `Component` or even a class `Class`. The possibility to inherit all properties from classes exists also in ArchJava where a component class can inherit a standard Java class or another component class.

The situation is different in ADLs, since specification languages inherently do not contain implementations, behavioral inheritance cannot be used. Thus, the inheritance can usually be used only for two purposes: creating conceptually specialized hierarchies (then inheritance is typically equivalent to the sub-typing) and for the structure reuse. This limited use of inheritance in ADLs is the main reason why it is implemented either insufficiently or not at all [Opluštil, 2003].

For example, Rapide allows its interface types to inherit from other types by using OO methods, resulting in structural sub-typing. ACME also supports structural sub-typing via its extends feature. Fractal ADL enables users to define new component type as an extension of an existing type and to override a component definition, but not to specialize inherited bindings.

SOFA CDL[6] uses the *frame* term for component types. One frame can inherit from another frame and then port declarations are reused. To compose several frames, SOFA introduce *architecture* construct, where an architecture implements a frame and may inherits from an another architecture. In this way, internal components and connections are reused. Ports are specialized using interface redefinition i.e. by the following statement `frame ComponentName inherits InheritedCompName changes InterfaceInstance1:: OriginalInterfaceType1 => NewInterfaceType1`. Specialization of inherited connections is supported by the statement: `newTie1, newTie2 replacing originalTie subsume subcompInstName:intInstName to intInstName exempt: subcompInstName:intInstName`.

Other ADLs do not specify inheritance between descriptors, they usually use inheritance uniquely for creation of sub-interfaces. In UML, the component entity inherits from the structured class entity and therefore they can participate in generalization relationship in the same way as classes do. In C2 [Medvidovic *et al.*, 1997] it is possible to define subtypes of all internal building blocks. Darwin [Magee *et al.*, 1995] can derivation descendants from one or more interface types. Rapide [Luckham *et al.*, 1995a] and its sublanguage for types allows deriving a new interface type by inheritance, including the capability of dynamic substituting of sub-types for super-types; inheritance of higher levels (Architecture, ...) is not supported. In xADL 2.0 [Dashofy *et al.*, 2001] provides single inheritance for type extensions, and introduces some artificial dependencies among schemas. Wright [Allen et Garlan, 1994] use connector as composition patterns among components.

If we exclude component frameworks which implement components by use of OOP and its class-based inheritance, there only component-oriented languages support behavioral inheritance. In the rest of this section we compare how related COLs integrate inheritance aspects such as: the structure inheritance , the behavior inheritance and abilities to extend and specialize particular definitions in a component descriptor (*i.e.* ports, internal components and connections.)

As related COLs we consider ACOEL [Sreedhar, 2002], ArchJava [Aldrich *et al.*, 2002], CLIC [Bouraqadi et Fabresse, 2009] and CompJava [Schmid et Pfeifer, 2008], because these languages combine implementation and architecture specification. We do not compare with ComponentJ [Seco et Caires, 2000 ; Seco *et al.*, 2008] and Bichon [Xu et Ren, 2010].

ComponentJ is an inheritance-free language where authors prefer to avoid inheritance in favor of object composition. ComponentJ emulates standard object-oriented implementation inheritance by feedback connection to the component itself and method calls forwarding. Authors show that the *initial receiver lose* problem can be solved using additional required port ("self" port) and feedback connections to handle the context. This solution increases rapidly the complexity of the system. In larger systems, where hierarchical concept modeling is used, code becomes difficult to maintain.

Bichon defines composition-oriented components for reuse. The authors analyze interaction between classes, which occurs via inheritance. Their reuse system is based on the observation that super-class needs to provide declarations, which sub-class requires and its consequence saying that when overriding occurs, a sub-class provides declarations which a super-class requires. This bi-directional interaction is performed via the Bichon's *mixinner* interface. Rather then seeing inheritance as a bi-directional interaction between classes describing components, we try to apply the widely accepted semantics of inheritance in the component description context

[6] CDL means Component Definition Language

Structure inheritance is partially supported in all related COLs. We say partially, because CompJava do not allow the reuse of internal components and connections specification. Ports declarations can be reused via component type definition. Except that they use a different terminology, the languages define component type as a set of port names including interface references and roles specification. And then a component type can be defined as an extension of an existing component type

Behavior inheritance is fully supported only in CLIC and ArchJava languages. ACOEL model supports implementation inheritance by the *extend* statement, but a child cannot access any of the internals (implementation classes, methods) of a parent, except via the input ports of the parent, *i.e.* `this.<portname>.<servicename>` (composition-like approach). The advantage of this black-box approach is that it preserve encapsulation of parent components. We support white-box approach to be able to specialize services implementations which are not provided by a parent.

Ports specialization is not supported in ArchJava, because adding new provided methods to an existing port might cause ambiguities if these provided methods were required by a connected component, and provided by a different component. There would then be two components providing the same required method, breaking ArchJava's connection rules. Adding required methods to an existing port would make the component class non-substitutable for the component super-class, because connections made to the super-class might not provide the sub-class's required methods. Required methods in a new port are also problematic, because the new port might not be connected at all.

Ports extension is well supported. CLIC model does not support additional provided port, because this model allows components to have only one provided port. The idea of a single provided port is based on the observation that developers do not know beforehand, which services will be specified by each required port of a client component. Therefore it is hard to split component functionality over multiple ports. We see this as a unnecessary limitation of modeling power.

On the other hand, in the CompJava, a component type may extend another component type and it inherits all ports. It may extend the interface of inherited provided ports or may add provided ports. Extension of required ports is not allowed due to the substitutability policy of the CompJava model.

Architecture extension and specialization. ACOEL and ArchJava treat internal components as regular instance variables of classes and therefore there is no way to specialize inherited internal components. CompJava supports inheritance of component types only. Component types do not involve internal components and connections declarations, therefore architecture cannot be reused.

Substitutions. ACOEL, ArchJava and CompJava define sub-type relation as defined by Liskov [Liskov et Zilles, 1974]. In general, a component type is a sub-type of another one if it provides at least the same and requires at most the same. To ensure ACOEL use a type system checking. CompJava and ArchJava forbid additional requirements (in inherited types) and then they restrict substitutability by the sub-type relation.

In Table 4.1, we make a summary of this comparison.

	CBPLs				
criterium/model	ACOEL	ArchJava	CLIC	CompJava	COMPO
Structure inheritance	yes	yes	yes	yes	yes
Behavior inheritance	yes	yes	yes	no	yes
Addition of:					
Provided ports	yes	yes	no	yes	yes
Required ports	yes	yes	yes	no	yes
Internal components	yes	yes	yes	no	yes
Connections	yes	yes	yes	no	yes
Specialization of:					
Provided ports	yes	no	yes	yes	yes
Required ports	yes	no	yes	no	yes
Internal components	no	no	yes	no	yes
Connections	no	no	yes	no	yes
Substitution					
with additional requirements	no	no	yes	no	yes

Table 4.1 : Comparative table of inheritance in related COLs

4.9 Summary

In this chapter, we have proposed an original and complete inheritance system for a component-based programming language. In the beginning, we have motivated the need for an inheritance system in a COL by showing cases where an inheritance mechanism is inevitable for reusing structural definition of descriptors. Then we have specified what are the subjects for inheritance in case a component-oriented language. For each identified subject we discussed the impact of extending descriptors and specializing inherited subjects together with the reasons which led us to incorporate this particular solution.

We believe that COMPO inheritance mechanism offers a reasonable solution for structural and behavior reuse, because our solution does not make separation of modeling (designing architectures) and implementation (designing behavior). The system promotes reuse over type-safe substitution, and thus enables developers to create new component descriptors by specializing and extending existing descriptors without a lot of constraints. Existing object-oriented programming languages that succeeded in the few last decades, prove the inheritance usefulness and practicalness, because it has been used extensively in existing applications. Indeed, developers are much more interested in specializing and extending at the same time provisions and requirements of a component, and less on substitutability, which they can manage manually (by satisfying additional requirements, if needed).

We have shown that descriptor-based inheritance is useful for both development-for-reuse and development-by-reuse. Indeed, on the one hand, it allows developers to build new component descriptors, to be put in component libraries, by inheritance links between descriptors. On the other hand, developers can build their applications by extending or specializing existing component descriptors. They can thus inherit existing architectures and capitalize on good designs where well-

established architecture styles or patterns are applied.

We conclude this chapter by recalling the definitions and the choices that we have made in this chapter.

4.9.1 Definitions made

Sub-descriptor A descriptor may extend and specialize another descriptor, such a descriptor is then called a *sub-descriptor*. (*cf.* 16)

Super-descriptor If a descriptor C is defined as a sub-descriptor of a descriptor D, then we say that D is a *super-descriptor* of C. (*cf.* 17)

The service lookup mechanism When a provided port receives a service invocation, it treats the invocation according to the Algorithm 3. In the case of lookup, the port looks up the requested service in the descriptor of the component that owns the port. If this descriptor does not implement the service, the port asks its super-descriptor, and so on, up the inheritance chain. When the services is found, the arguments are connected to the parameters ports of the service, and the service is executed. When the service is not found, then the `doesNotUnderstand` service of the port is executed. (*cf.* 19)

4.9.2 Choices made

Choice 23 Port descriptions, architectures (connection descriptions sets) and services are subjects of COMPO inheritance

Choice 24 The inheritance mechanism in COMPO is single inheritance.

Choice 25 A sub-descriptor inherits all subjects of its super-descriptor (its parent), *i.e.* all ports descriptions, the architecture and all services definitions.

Choice 26 A sub-descriptor can introduce new services and its instances can invoke, redefine and specialize services defined by its super-descriptor.

Choice 27 Every sub-descriptor has, by default, the `super` internal provided port. Service invocations sent through this port are looked up starting from the super-descriptor of the descriptor owning the service in which code the service invocation are emitted.

Choice 28 A sub-descriptor can introduce a new provided port description.

Choice 29 A name of a newly added port of a sub-descriptor cannot clash with existing port names (inclusive inherited port names).

Choice 30 It is not possible to specialize the visibility nor the role of an inherited port description in a sub-descriptor.

Choice 31 A sub-descriptor can specialize the list of service signatures (the interface) of an inherited port description.

Choice 32 A sub-descriptor can introduce a new external required port description or it may specialize the interface of an inherited description of an external required port.

Choice 33 A sub-descriptor inherits the architecture of its super-descriptor. It may introduce new internal required ports descriptions and new connection descriptions. It may specialize the inherited connection descriptions and the interfaces of inherited internal required ports descriptions.

CHAPTER 5

Integrating reflection

One can and should "open languages up," allowing users to adjust the design and implementation to suit their particular needs.
Gregor KICZALES.

Preamble

In this chapter we describe a meta-model which equips COMPO *with reflection capabilities making it possible to achieve architecture reasoning, and static or run-time model and program transformations, all within the context of one language. In sections 5.3 and 5.4, we propose the meta-model architecture. Sections 5.5, 5.6 and 5.7 describe how we have reified concepts:* component, descriptor, port *and* service *in order to make them accessible in* COMPO *programs. In the end of the chapter we discuss related work.*

5.1 MDE, the motivation for reflection

MODEL Driven Engineering (MDE) raises the level of abstraction of artifacts in the development life cycle by shifting its emphasis from code to models and model transformations. According to the separation of concerns principle, MDE advocates the isolation of business concerns from their technical achievement. The idea is that the business concerns can be modeled independently from any platform concerns. Therefore, business models are not corrupted by technical concerns. In this way, the main part of the development becomes an activity upstream, dedicated to business aspects through the elaboration of the application model that abstracts away technical details [Blanc *et al.*, 2007]. These primary models are then enhanced into final software products by a chain of transformations. MDE provides a method for developers to master these complexities by both separating concerns and systematically describing the design, implementation and validation processes of development [Terrier et Gérard, 2006].

However, when applying an MDE process, different languages have to be learned and mastered to design and develop a final solution, *e.g.* an ADL for the architecture design, a programming language for the implementation (model transformations only generate skeleton implementations), a language for expressing architecture constraints (such as OCL) and possibly a language for model transformations. This problem of a missing *conceptual continuum*, similar to the continuum that exists between object-oriented design and implementation [Muller *et al.*, 2005; OMG, 2011c], makes it difficult to apply MDE techniques and processes in a straightforward way, because it requires the experts from different domains (constraints, transformations, etc.) to cooperate, which is not always an easy task.

Indeed, the continuum encompasses the activity of writing all kinds of meta-programs. This globally means to allow software engineers to achieve, using the same language defined by a unique component-based meta-level `M`, not only applications (architectures and code) but also all those meta-programs, *e.g.* constraint-checking or model transformation or program transformation programs, that use or manipulate `M` constitutive elements and their instances, either statically or at run-time.

It appears that a reflective component-oriented programming and modeling language is a possible original solution to such a requirement. A reflective language or system provides a principled (as opposed to ad hoc) means of achieving open engineering [Blair *et al.*, 1998]. We believe that a reflective component-oriented language is a way to do MDE in the context of component, similarly to reflective OOP languages make it possible to do MDE for object-oriented designs and programs. Reflection enables language users to reason about architectures, to perform model transformations, to examine and modify the structure and behavior of entities (components, in our case) at run-time. Such a language contributes to solve the above issues by having the same description of architectures at design and run-time.

In this chapter we present entities involved in the component-based development integrating reflection capabilities. In the core of this chapter, we reify, following the idea of "*everything is a component*", the core component concepts to build up an executable meta-model, allowing introspection and intercession on programs elements and their instances. The meta-model can be used at all stages of component development to manipulate standard and "meta"-components as first-class entities.

Using such a language opens the possibility that architectures, implementations and transformation can all be written at the component level and possibly (but not mandatorily) using a unique language.

5.2 Reflection & Reification

Because some readers may not be aware of the field and vocabulary, this section provides a short introduction. Experts can omit it.

Reflection is a concept arised from the artificial intelligence field, as the ability of a system to *reason about and act upon itself*. Reflection is about meta-computation, *i.e.* computation about computation. This was considered as an emergent property responsible for intelligent behavior [Costa Soria, 2011]. More then 30 years passed since Smith introduced the reflection concept in his doctoral dissertation [Smith, 1982]. Meanwhile, reflection become popular and spread to other fields, such as object-oriented systems [Cazzola, 1998], middleware [Kon *et al.*, 2000; Costa *et al.*, 2006], software architectures [Cuesta *et al.*, 2002] or dynamic petri nets [Capra et Cazzola, 2009]. This was mainly thanks to the contributions of Pattie Maes, who contributed to summarize the existing notions about reflection:

> "*Computational Reflection is the activity performed by a computational system when doing computation about (and by that possibly affecting) its own computation.*
>
> *A reflective system is a computational system which is about itself in a causally connected way.* " [Maes, 1987]

Reflective systems are generally structured in two levels: the *base-level* and the *meta-level* [Costa Soria, 2011].

The **base-level** provides the system's functionality. It defines a computational system (*i.e.* a set of computational elements) that reasons about and acts upon some part of the world, usually called the domain of the system. This level incorporates internal structures representing the domain and a program prescribing how these structures may be manipulated.

On the other hand, the **meta-level**[1] provides the reflective capability. It defines a computational system that reasons about and acts upon another computational system, *i.e.* the defined in the base-level (the base-system). Thus, it incorporates structures representing the base-level and a program that manipulates and changes such structures.

Both levels, the meta-level and the base-level, are **causally connected**: the structures defined in the meta-level and the domain they represent (*i.e.* the base-level) are linked in such a way that if one of them changes, this leads to a corresponding effect upon the other [Maes, 1987]. In other words, the changes performed by the meta-level on its data structures are reflected somehow in the real system, *i.e.* the base-level. In order to observe or change something, this must be represented in a way that a program can manipulate it.

[1] In general, the term meta- refers to an artifact that reasons and acts upon another artifact. For instance: a meta-component is a component that acts upon components.

This is addressed in the notion of **reification**, which is the action of exposing the internal representation of a system in terms of programming entities that can be manipulated at run-time. The opposite process is called reflection[2], which effects the changes made to the reified entities into the system. There are many definitions of reification, like [Malenfant *et al.*, 1996], but these definitions may induce to confusion since they do not refer to the base-level and meta-level a reflective system is characterized by. This is taken into account in the definition from Carlos E. Cuesta:

> "*Reification is the process that shifts-up an artifact from the base-level to the meta-level, where this artifact will be manipulated. Reflection is then the inverse process that shifts-down the artifact from the meta-level to the base-level. Thus, the reification and reflection processes implement the causal connection among the base-level and the meta-level.*" [Cuesta *et al.*, 2002]

Finally, there are two kinds of operations that can be performed at the meta-level: *introspection* and *intercession*. Introspection is the ability of a program to observe, and thus reason, about itself. That is, it comprises the operations of a program defined at the meta-level which examines the data structures and program operations of the base-level. Intercession is the ability of a program to modify its execution state. That is, it comprises the operations of a meta-level program which change the data structures and program operations of the base-level (see Figure 5.1).

[2]Some authors prefer not to use the term reflection to define the action of reflecting changes on the base-level, and use the term absorption instead. The reason is to avoid confusions with the global term of Reflection. However, we prefer the use of this term to preserve the symmetry of operations, in accordance with [Cuesta *et al.*, 2002].

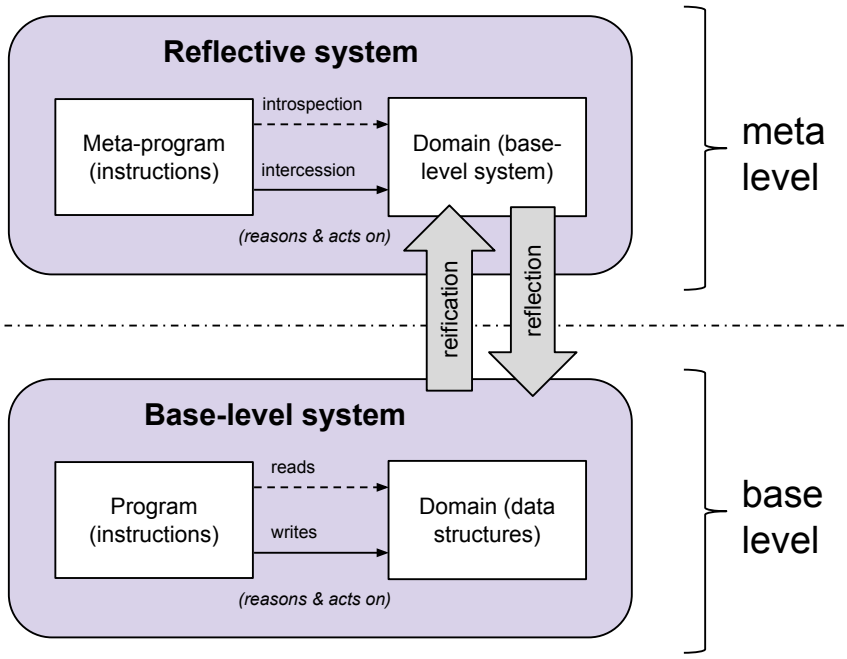


Figure 5.1 : Reflection of software systems [Costa Soria, 2011]

5.3 Requirements for the meta-model architecture

In accordance with COMPO's philosophy (*see* Section 3.1), which states that COMPO should be simple, minimal and uniform as much as possible we need a meta-model architecture with the following requirements:

Everything is a component to access the reified concepts of COMPO (descriptors, ports , connection and services) from COMPO itself. This will make it possible to write transformations and verifications of COMPO applications in COMPO. For example, our previous work [Tibermacine *et al.*, 2011] shows that architectural constraints can be successfully realized as components. The idea is attractive when it comes to verification of architectures quality attributes, especially, after an evolution was performed. In COMPO, a descriptor describes the architecture of its instances and therefore constraints should be evaluated on descriptors. If we want to apply constraints realized as COMPO components on an architecture, we need to be able to connect a descriptor to a constraint component. Consequently, because connections are connecting components (*i.e.* instances of descriptors), descriptors should be reified (realized) as components.

Explicit meta-descriptors to model new kinds of descriptors, ports or constraint components. The ability to design new meta-descriptors opens up the language. The language can be then adapted to ones needs because it is possible to extend and customize the core concepts of the language. For example a new communication protocol can be designed by creating a new kind of port, as we show in Section 5.6.

Among reflective class-based object-oriented languages (like SMALLTALK [Goldberg et Robson, 1989] or CLOS [Bobrow *et al.*, 1986]), ObjVlisp [Cointe, 1987] provides **minimal** (only two classes) self-described definition of its reflective architecture which makes it possible to satisfy these requirements (in the OO context). The architecture is based on only five postulates (listed below) and is really minimal, because there is only one kind of object: a class is an object and a meta-class is a class that creates classes. In other words, there is no distinction between classes and instances. The only sole difference is the ability to respond to the creation message: *new*. Only a class knows how to deal with it. A meta-class is only a class that can create new classes.

ObjVlisp in 5 postulates [Cointe, 1987]

P1 : object = <data, behavior>

P2 : Message passing is the only means to activate an object

P3 : Every object belongs to a class that specifies its data (slots or instance variables) and its behavior. Objects are created dynamically from their class

P4 : Following P3, a class is also an object therefore instance of another class its meta-class (that describes the behavior of a class).

P5 : A class can be defined as a sub-class of one or many other classes.

The postulates introduce an **infinite recursion**: *A class is an object and therefore it is an instance of another class (its meta-class) that is an object too (instance of a meta-meta-class) that is an object too (instance of another meta-meta-meta-class), etc.* To stop this infinite recursion, ObjVlisp defines `Class` as both the initial class and meta-class. `Class` is an **instance of itself** and all other meta-class are its instances, as shown in Figure 5.2.

We adopt the reflection architecture of ObjVlisp for the component-based context and we build our meta-model by reifying descriptors, ports and services as components. In the following, we refer this as the **component-oriented reification**.

Choice 34 *Descriptors, ports and services are true components, instances of descriptors* DESCRIPTOR, PORT *and* SERVICE *respectively.*

The reason why we do not reify connections is given in Section 5.6 .

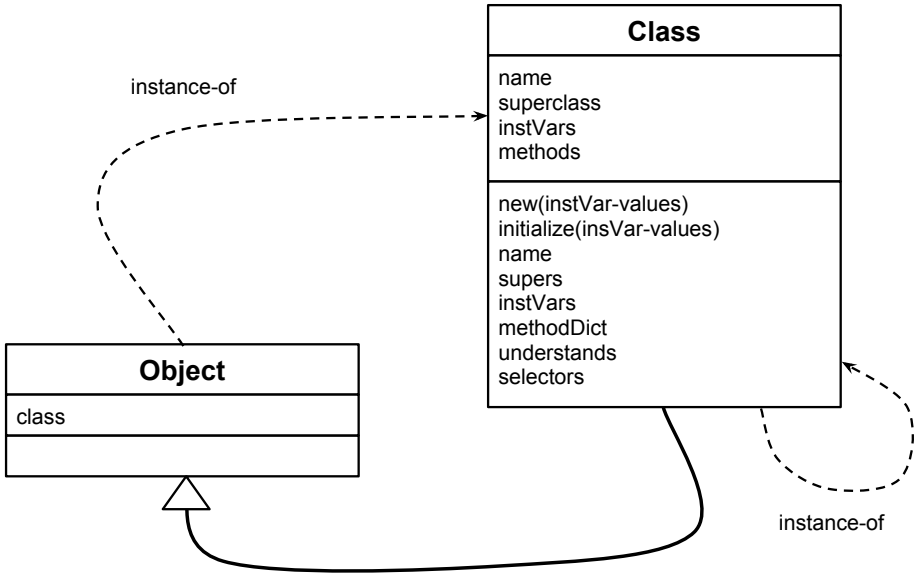


Figure 5.2 : ObjVlisp `Class` is an instance of itself to solve the infinite recursion of the 5 ObjVlisp postulates.

5.4 The meta-model

This section introduces an adaptation and extension of COMPO's model presented in Chapters 3 and 4 to allow for structural reflection, *i.e.* "*to provide a complete reification of both a program currently executed as well as a complete reification of its abstract data types*" [Demers et Malenfant, 1995]. At this point we do not deal with behavioral reflection, although, our global solution makes it possible to define new kind of ports (*see* the example later on in this chapter) in which service invocation can be modified. This is a very limited kind of behavioral reflection.

The MOF model presented in Figure 5.3 describes how its elements, representing the component-level concepts introduced in COMPO, are organized (inheritance and instance-of relations), as we reify them as first-class entities accessible in COMPO's programs. To keep our contextual component-level terminology: all elements in Figure 5.3 are descriptors. Our reflective architecture is based on the two following choices:

Choice 35 *Descriptor* COMPONENT *is a basic descriptor and the root of inheritance tree, all descriptors inherit it*

Choice 36 *Descriptor* DESCRIPTOR *is a sub-descriptor of descriptor* COMPONENT. *It describes descriptors (it is a meta-descriptor) and is the instance of itself.*

Descriptor is the descriptor of descriptors (it resembles to class `Class` of ObjVlisp), all descriptors are instances of it. All descriptors inherit from *Component*, except *Component* itself which is the root

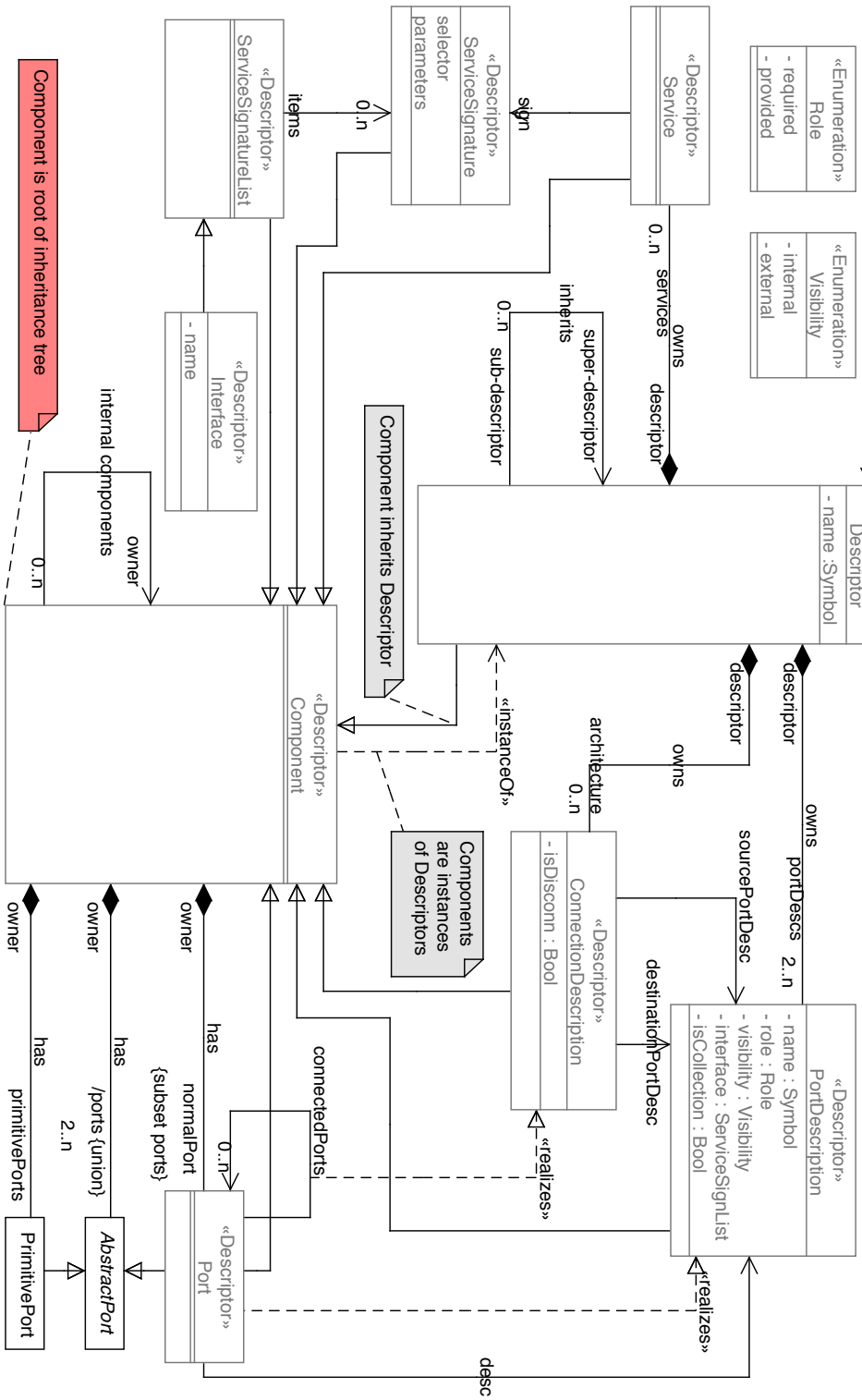


Figure 5.3 : The meta-model of Compo showing the integration of reflection. All elements, except the primitive ports, in the scheme were reified as COMPO descriptors. The grayed color denotes original concepts shown in Figure 3.2.

of the inheritance tree. All descriptors are components. *Descriptor* is instance of itself, it is its own descriptor. This solves at the model level the infinite regression on descriptions (similar to the one of ObjVlisp 5 postulates).

Our modeling scheme to represent descriptors as components conforms to the MOF solution for reflection:

> "*Reflection introduces Object as a super-type of Element in order to be able to have a Type that represents both elements and data values. Object represents 'any' value and is the equivalent of java.lang.Object in Java.*" [OMG, 2011a].

Component in Figure 5.3 conforms to `MOF::Reflection::Object`. *Descriptor* conforms to `UML::Classes::Kernel::Classifier`.

The following sections describe the COMPO's reflective implementation of the main meta-model elements of in Figure 5.3. The sections present the associated language constructs and give some primary examples of their use. Each element of our meta-model is implemented as a COMPO descriptor. The inheritance relations in the meta-model are directly implemented in COMPO using its descriptor-level inheritance system and its ability to create sub-descriptors of descriptors.

5.5 First-class descriptors and components

The conceptual meta-model in Figure 5.3 and its simplified excerpt in Figure 5.4 show the two basic concepts: *component* and *descriptor*. The component-oriented reifications of those concepts are represented by descriptors COMPONENT and DESCRIPTOR. The descriptor DESCRIPTOR inherits from the descriptor COMPONENT, which makes any descriptor a component, and where DESCRIPTOR and COMPONENT are descriptors. The fact that DESCRIPTOR is an instance of itself solves the potential infinite regression induced by the need for anything to have a descriptor.

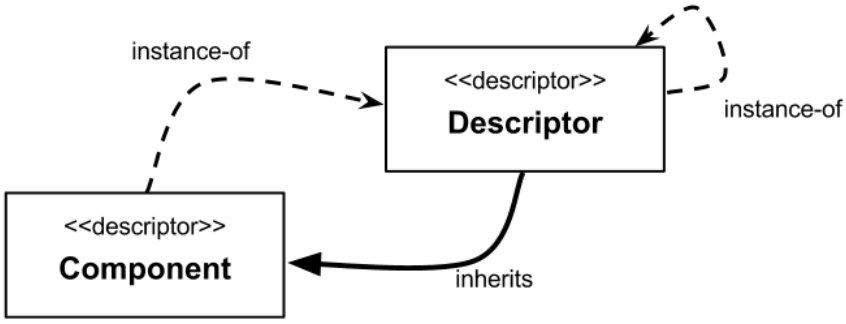


Figure 5.4 : Excerpt of the meta-model (*see* Figure 5.3) showing the two basic elements: *component* and *descriptor* with their relations.

COMPONENT defines the basic structure and behavior shared by all components. Definition in Listing 5.1 shows that all have an external provided port named `default` described by the *univer-*

```
Descriptor Component {
  provides {
   default : { getPorts(); getPortNamed(name);
             getDescriptor(); getOwner();
             getIdentityHash();
            }
  }
  requires { args[] : * }
  internally requires {
    super : * ofKind SuperPort;
    self : * ofKind SelfPort;
  }
  service getPorts() {...}
  service getPortNamed(name) {...}
  service getDescriptor() {...}
  service getOwner() {...}
  service getIdentityHash() {...}
}
```

LISTING 5.1 : The COMPONENT descriptor.

sal interface ∗ (*see* Definition 5), an external required collection port `args[]` to connect arguments (*see* Section 3.3.2) and two internal provided ports named `self` and `super`. The `self` port allows a component to invoke its own services. Service invocations sent through the `super` port are looked up starting from the super-descriptor of the descriptor owning the service in which code the service invocation is emitted (see Chhoice 27).

Our meta-model enables users to define new kinds of ports by creating sub-descriptors of descriptor PORT (*see* Listing 5.5). In order to be able to define the type of a port in its declaration in a descriptor, we choose (*cf.* Choice 37) to provide an operator **`ofKind`**.

Choice 37 *A port declaration can be made more specific by putting the following statement* **`ofKind`** *`<descriptor>` after interface specification. The statement specifies that a port will be created as an instance of the specified descriptor `<descriptor>`.*

For example, in case of the `self` and `super` ports, the `ofKind SelfPort` and `ofKind SuperPort` statements specify that these ports will be created as instances of descriptors SELFPORT and SUPERPORT respectively.

Descriptor COMPONENT defines the four following services:

- services `getPorts()` and `getPortNamed(name)` return all (external and internal) ports (resp. a particular port) owned by the receiver. When these services are invoked via the `super` port, they return port(s) owned by a component, which are defined by the super-descriptor of the descriptor of the component. For example for a component defined with the following descriptor:

  ```
  Descriptor A { requires { reqA : { ser1() } } }
  ```

An invocation of service `getPorts()` sent through port `default` returns a collection component referencing ports `default`, `self`, `super` and `reqA` of an instance of descriptor A. The service return the same result if it is invoked through port `self`. In case of invocation via port `super` the service returns only references to ports `default`, `self` and `super`, because only those are defined by the super-descriptor (COMPONENT) of descriptor A.

- service `getDescriptor()` returns the receiver's descriptor.
- service `getIdentityHash()` returns the primitive (VM) identity hash of components.
- service `getOwner()` returns the owning component of the receiver or null if the receiver is not an internal component

```
Descriptor Descriptor extends Component
{
  internally requires {
    name : Symbol; /* an identifier string */
    ports[] : PortDescription;
    architecture[] : ConnectionDescription;
    services[] : Service;
  }
  service getPorts() {...}
  service getPortNamed(name) {...}
  service getDescriptor() {...}
  service getOwner() {...}
  service getDescribedPorts() {...}
  service getDescribedConns() {...}
  service getService(selector, arity) {...}

  service new() {...}
  service newNamed(name,superDesc) {...}

  service addService() {...}
  service removeService(selector,arity) {...}

  service addPortDescription(pd) {...}
  service removePortDescription(pd) {...}

  service addConnDescription(cd) {...}
  service removeConnDescription(cd) {...}
}
```

LISTING 5.2 : The DESCRIPTOR descriptor.

Listing 5.2 shows COMPO definition of DESCRIPTOR. Its definition states (*cf.* Listing 5.2) that all descriptors have, in addition to what is defined in COMPONENT, four internal required ports:

- `name`,

- `ports[]`, a descriptor has a collection of port's descriptions (instances of PORTDESCRIPTION, *see* Listing 5.3) according to which ports of its instances will be realized.
- `architecture[]`, a descriptor has a description of its instances internal architecture in the form of a collection of connection's descriptions (instances of CONNECTIONDESCRIPTION, *see* Listing 5.4) according to which connections will be realized.
- `services[]` to store the collection of services of its instances.

DESCRIPTOR defines services for instance creation. The service `new` implements the instantiation mechanism (*cf.* Section 3.3.1) of COMPO and the `newNamed(name, superDesc)` service makes it possible to create new descriptors. There are several services for introspection (various read-accessors such as `getDescribedPorts()`) and for intercession (such as `addService(service)`). These services, together with those inherited from COMPONENT set the basis for creating more complex reflective operations.

```
Descriptor PortDescription extends Component{
  provides {
    default : { setName(name); getName(); setRole(role); getRole();
                setKind(kind); getKind(); setInterface(intf);
                getInterface(); setVisibility(vis); getVisibility();
                isCollection(); setIsCollection(bool);
              }
  }
  internally requires {
    name : Symbol;
    role : Symbol;
    visibility : Symbol;
    interface : Interface;
    kind : Symbol;
    isCollectionPort : Bool;
  }
}
```

LISTING 5.3 : The PORTDESCRIPTION descriptor.

Meta-descriptors provide a definition of the descriptors. An important benefit of first-class status of descriptors is customization of the descriptors behavior [Ledoux et Cointe, 1996], *i.e.* the ability to assign properties to descriptors (*e.g.* being abstract, being safely-substitutable, supporting multiple inheritance), independently from the base-level code. Because descriptors have the ability to manipulate their own structures, they can implement a program introspection. Consequently, meta-descriptors support a circular definition of the system reducing the boundary between users and implementors. In the rest of this section we present basic examples of using introspection, intercession and an example of a new kind of meta-descriptor.

Figures 5.5 and 5.6 show diagrams of component-based reifications of the component and descriptor concepts.

```
Descriptor ConnectionDescription  extends Component {
  provides {
    default : { getSourceComponent(); getSourcePort(); getDestinationComponent();
                getDestinationPort(); getKind(); setSourceComponent(scd);
                setSourcePort(spd);  setDestinationComponent(sdc);
                setDestinationPort(sdp); isDisconnection();
                setIsDisconnection(bool);
              }
  }
  internally requires {
    sourceComponent : Symbol;
    sourcePort : Symbol;
    destinationComponent : Symbol;
    destinationPort : Symbol;
    isDisconnection : Bool;
  }
}
```

LISTING 5.4 : The CONNECTIONDESCRIPTION descriptor.

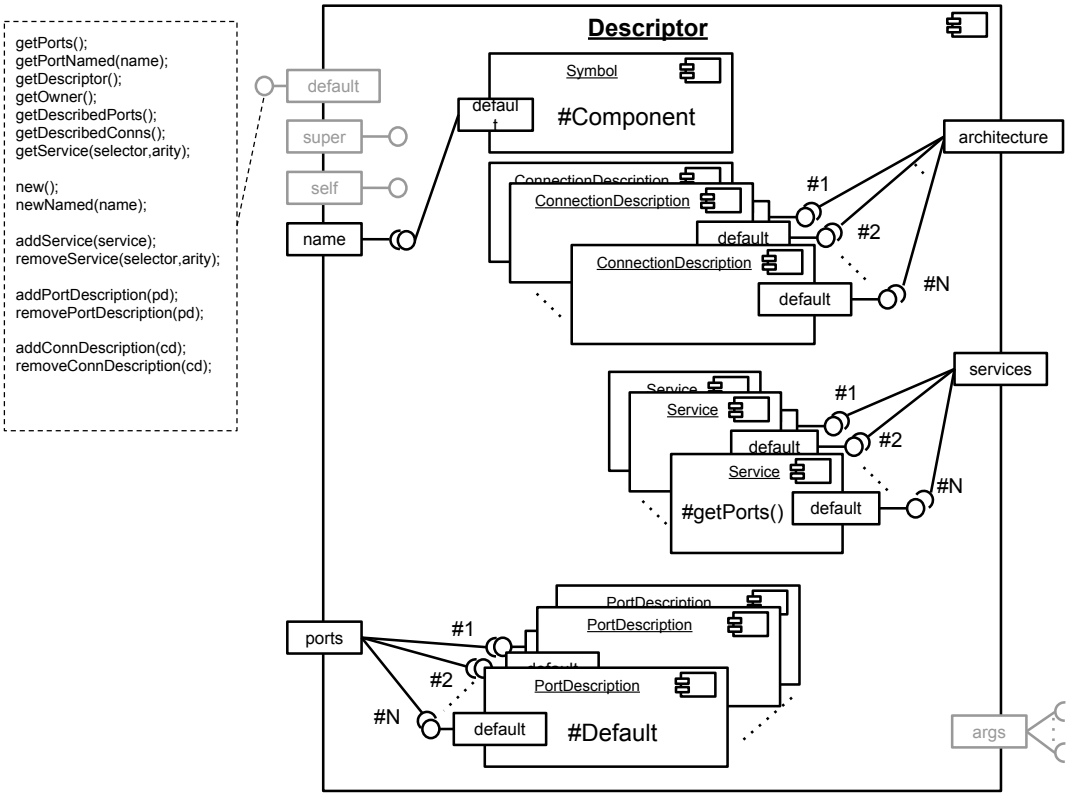


Figure 5.5 : A diagram of a component-based reification of the component concept. Greyed parts denote inherited parts.

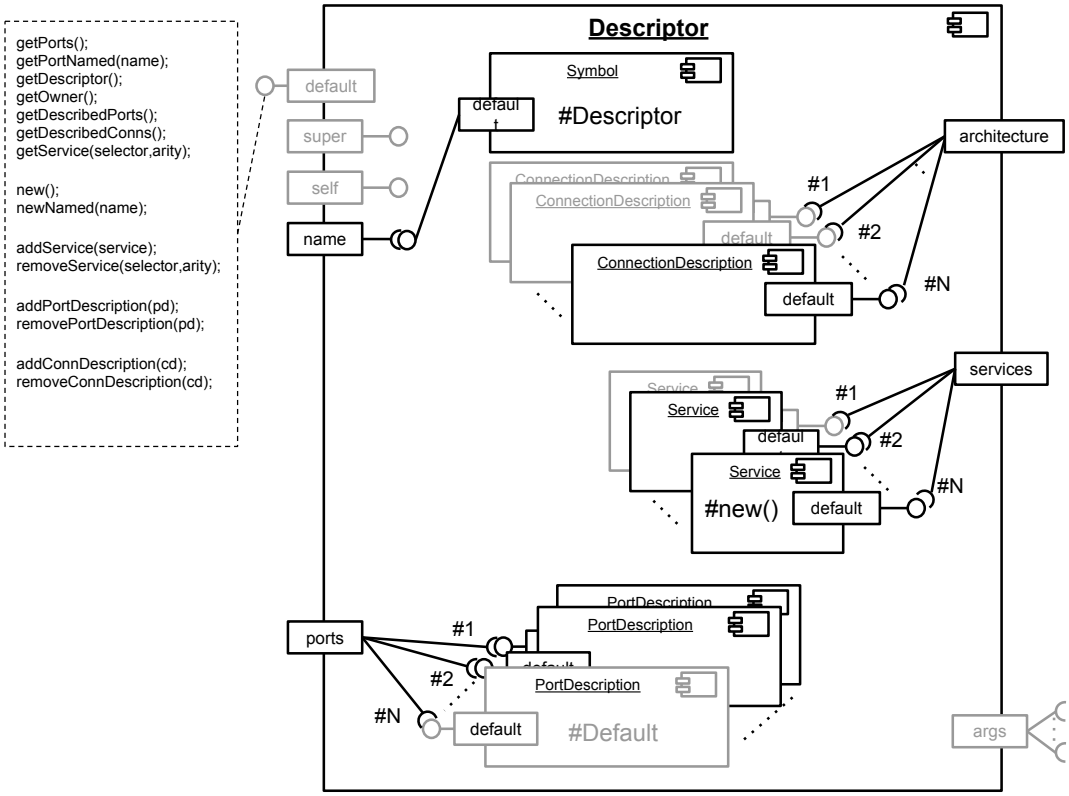


Figure 5.6 : A diagram of a component-based reification of the descriptor concept. Greyed parts denote inherited parts.

An introspection example The following code snippet shows a basic use of introspection. The expression returns the descriptions (*i.e.* instances of descriptor PORTDESCRIPTION, *see* Listing 5.3) of ports `default`, `self` and `super`, which are defined by the descriptor COMPONENT, see Listing 5.1.

```
Component.getPortNamed('default').getDescribedPorts();
```

An intercession example The following code snippet shows the descriptor (named SERVICEMOVER) of a refactoring component, which combines `get`, `remove` and `add` services to move a service from one descriptor to another.

```
Descriptor ServiceMover {
  requires {
    srcDesc : Descriptor;
    destDesc : Descriptor
  }
  service move(selector) {
    |srv|
    srv := srcDesc.getService(selector);
    destDesc.addService(srv);
    srcDesc.removeService(selector);
  }
}
```

An example of defining a meta-descriptor DESCRIPTOR is a meta-descriptor. A new meta-descriptor can be defined by extending it. As an example, consider the following issue. Having an inheritance system, it is possible for a sub-descriptor SD to define new required ports, thus adding requirements to the contract defined by its super-descriptor D. In such a case, the substitution of an instance of D by an instance of SD needs specific checking (child-parent incompatibility problem 4.6 of inheritance systems in the component-based context). It may be wanted to define some descriptors that do not allow their sub-descriptors to add new requirements. Such a semantics is achieved by the DESCRIPTORFORSAFESUBSTITUTION definition shown in the following code snippet. The meta-descriptor extends the descriptor DESCRIPTOR and specializes its service `addPortDescription`, which implements the capability to add a port description. The service is redefined in a way that it signals an exception each time it is tried to add a description of an external required port.

```
Descriptor DescriptorForSafeSubstitution extends Descriptor
{
  service addPortDescription(portDesc) {
    | req ext |
    req := portDesc.isRequired();
    ext := portDesc.isExternal();
    if (reg & ext)
    { error('no new reqs. allowed'); }
    else { super.addPortDescription(portDesc); }
  }
  ...
}
```

An instance (a new descriptor) of the DESCRIPTORFORSAFESUBSTITUTION meta-descriptor named TESTDESCRIPTOR extending descriptor COMPONENT could then be created by the following expressions:

- Run-time creation

  ```
  DescriptorForSafeSubstitution.newNamed('TestDescriptor', Component);
  ```

- Static creation

  ```
  DescriptorForSafeSubstitution TestDescriptor extends Component
  { ... }
  ```

To conclude this part on components and descriptors, let us show the PORTDESCRIPTION and CONNECTIONDESCRIPTION descriptors, *see* Listings 5.3 and 5.4 respectively. They simply declare a single provided port through which they offer getter and setter services for accessing the descriptor level descriptions of ports and connections. Such descriptions are useful to achieve static or dynamic architecture checking or transformation. In the case of a run-time transformation, a COMPO's implementation should ensure that these descriptor level descriptions and the internal representation of descriptor instances are causally connected. When the description changes, all instances should automatically be updated.

5.6 First-class ports

Generally the "port" concept is a higher-level abstraction of references known from OOP. Ports concern connections between components and communication, *i.e.* service invocations sent through them. They explicitly represent connection points and implicitly represent references. Their first-class status opens, in an explicit and simple way, the "door" to program-based manipulation of: (i) connection points, (ii) connections and (iii) service invocations. Thus, they make it possible to achieve scenarios similar to the ones in the object-oriented context, where first-class references are introduced [Arnaud *et al.*, 2010] or the ones where custom lookup objects are needed [Vraný *et al.*, 2012].

To reify ports are components is important for model verification and transformations and also to allow for defining new kinds of ports introducing new communications protocols. For example, a developer can create a new kind of port, which implements request/response protocol, *i.e.* for each service invocation sent via such a port, there should be a confirmation that the invocation was well received. It however induces two potential infinite regressions.

The first infinite regression is related to the definition: "*a port is a component having ports*". To solve the recursive nature of that definition we restrict the language capabilities by altering the definition in the following way: "*a port is a component having primitive ports*". The restrictions are captured by Definition 20 and Choice 38 introducing primitive ports:

Choice 38 *A port is a component and its ports are primitive ports.*

Definition 20 (Primitive port) *A primitive port behaves like a port but is not a component. Ports declared in the* PORT *descriptor (or its sub-descriptors) are automatically made primitive to avoid infinite regression.*

The second infinite regression is related to the fact that if ports are components, a component and one of its ports, should be connected via ports. To solve this, the attachment of a port to its owning component has to be primitive and in conjunction a language special construct is needed to provide access to a port seen as a component.

Similar issues would apply with first-class connections in the case where components are directly connected via ports. Having a solution where components are connected via theirs ports, we can consider connections between ports as primitive entities (references), and **we do not need to reify connections**. This entails no limitation regarding the capability to experiment with various kinds of connections [Mehta *et al.*, 2000] because our model makes it possible to define new kinds of ports and because of the capability it offers to put an adapter component in between any components.

The listing 5.5 shows the COMPO's definitions of the PORT descriptor that implements the *Port* concept and its sub-descriptor COLLECTIONPORT that implements collection ports. The descriptor PORT states that each port has:

- an *owner*, any port is owned by a component. A port is connected to its owner through external required port `owner`
- a set of *connected ports* (to which the port is connected) realized with external required collection port named `connectedPorts[]`
- a set of *delegated ports* (to which the port is delegated) realized with external required collection port named `delegatedPorts[]`
- a *name* symbol-component connected to the internal required port `name`
- an *interface* connected to the internal required port `interface`

PORT defines services for port introspection (e.g. `getName()`, ...) and port intercession (e.g. `connectTo(port)`, `invoke(service)`, ...). The difference between descriptors PORT and COLLECTIONPORT is the implementation of services `connecTo`, `disconnect` and `invoke`. Where COLLECTIONPORT descriptor manages the multiplicity of connections while PORT descriptor implements the services so that only one connection is possible.

```
Descriptor Port extends Component
{
  requires {
    owner : IComponent
    connectedPorts[] : IPort
    delegatedPorts[] : IPort
  }
  internally requires {
    name : IString;
    interface : IInterface
  }
  service getName(){...}
  service getIterface(){...}
  service invoke(service){...; <primitive_invoke>; ...}
  service isConnected(){...}
  service isDelegated(){...}
  service connectTo(port){...; <primitive_connectTo>; ...}
  service disconnect(){...}
}

Descriptor CollectionPort extends Port
{
  service invoke(service,index){...}
  service connectTo(port){...}
  service disconnect(index){...}
}
```

LISTING 5.5 : The descriptors PORT and COLLECTIONPORT

Services invocations are made via ports, for example the expression `printingPort.print('hello')`, where `printingPort` is a port of a component `c`, will invoke the service `print` of the component connected to `c` via `printingPort`. To use `printingPort` as a component, to send it a service invocation for example, requires a correct way to reference it, *i.e.* conforming to COMPO's meta-model and semantics. Such a correct way is to have a required port connected to the default provided port of `printingPort` seen as a component (*see* Figure 5.7.) To achieve this, we have introduced the **& operator**, for any port `p`, `&p` is such a required port.

Definition 21 (The & operator) *The & operator applied on a port, i.e. `&<portName>`, returns an on-demand created primitive internal required port, which is automatically connected to the `default` port of the component representing (reifying) port `<portName>`.*

Because primitive ports are not reified, the application of the & operator on a primitive port re-

turns itself, then a double application of the operator returns the same result as a single application, *i.e.* `&printingPort == &&printingPort`.

In the previous example, it is possible to write:

```
&printingPort.isConnected();
```

`&printingPort` is a primitive internal required port which is created on-demand (for performance reasons) and automatically connected to the `default` port (*see* Choice 10), itself a primitive port, of the `printingPort` port seen as a component. Invocations sent through such a port are invocations sent to the component representing the `printingPort` port. An example of use of the & operator is given in the next paragraph.

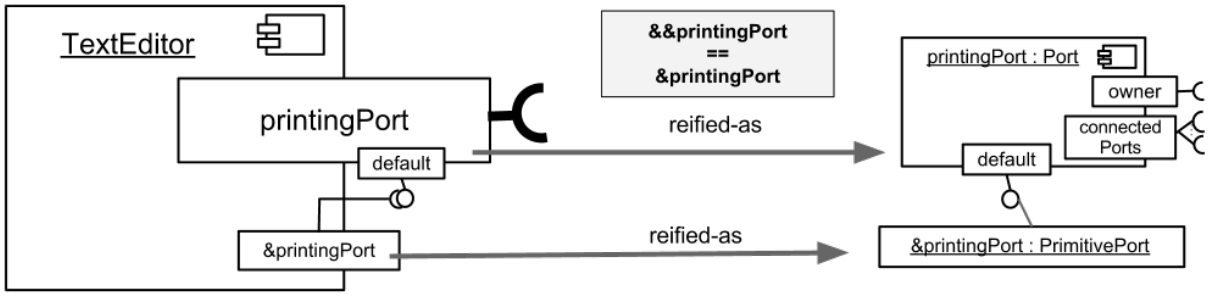


Figure 5.7 : The & operator for accessing the component-oriented reification of the `printingPort` port of an instance of descriptor TEXTEDITOR

Example: A new kind of port - an aspect port Listing 5.6 shows a toy integration of basic aspects to serve as an illustrating example. Descriptor ASPECTPORT defines in COMPO a new kind of required ports that have a special required port named `aspectComp`, to be connected to any component having `before` and `after` services, let's call such a component an aspect component. Descriptor ASPECTPORT redefines the standard service invocation semantics of ports so that the `before` and `after` services of the aspect component are invoked before and after the standard invocation.

The descriptor TEXTEDITOR shows (in Listing 5.7 a use of an aspect-port (note the **`ofKind`** operator to specify that an aspect required port is used).

Finally, we create an application (described by descriptor APPLICATION) which assemble an instance of TEXTEDITOR with a printer component, *see* Listing 5.7. The last **`connect`** statement in the architecture section of APPLICATION descriptor says that the `aspectComp` port of the aspect-port, here used as a first-class component accessed via internal required port `&textEd`, should be connected to the default provided port of a MYASPECTCOMPONENT, see Figure 5.8.

Example: A new kind of port - a read-only port Listing 5.8 shows the READONLYPROVIDEDPORT descriptor realizing a new kind of provided ports through which only services with no side effect on the architecture, *i.e.* services not affecting the internals of the component, could be invoked. It redefines

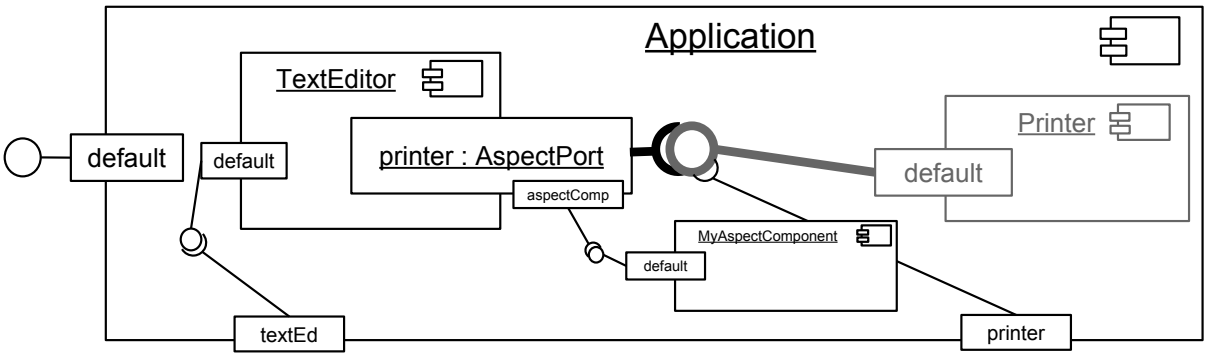


Figure 5.8 : The visualization of the use of an aspect-port

```
Descriptor AspectPort extends RequiredPort {
  requires { aspectComp : {before(); after(); }}
  service invoke(service) {
    aspectComp.before();
    super.invoke(service);
    aspectComp.after();}
}
```

LISTING 5.6 : The example of creating an aspect-port

```
Descriptor MyAspectComponent {
  provides { default : {before(); after()}}
  service before(){...}
  service after(){...}
}
Descriptor TextEditor {
  requires { printer : {print()} ofKind AspectPort  }
  ...
}
Descriptor Application {
  internally requires {
    textEd : TextEditor;
    printer : Printer;
  }
  architecture {
    connect textEd to default@(TextEditor.new());
    connect printer to default@(Printer.new());
    connect aspectComp@(&textEd) to default@(MyAspectComponent.new());
  }
}
```

LISTING 5.7 : The example of using an aspect-port

the standard service invocation to check whenever it is correct or not to invoke the requested service and it also redefines the standard connecting service in a way that a provided port of kind read-only can be delegated to only another read-only port.

```
Descriptor ReadOnlyProvidedPort extends ProvidedPort
{
  service invoke(service) {
    |bool1 bool2|
    bool1 := owner.implements(service);
    bool2 := owner.isConstantService(service);
    if(bool1.and([bool2]))
    { super.invoke(service); }
    else { ... }
  }
  service connectTo(port) {
    if(port.getDescriptor().isKindOf(ReadOnlyPort))
    { super.connectTo(port); }
  }
}
```

LISTING 5.8 : The example of creating and using an aspect-port

The explicit status of ports is a way to further control references between entities. For example, the above case of aspect required ports represents a way to realize a join point defined for all the users of a component having such a port. Or, the read-only example illustrates the fact that using different kinds of provided ports can facilitate different view-points on a component, in this case the read-only view-point.

5.7 First-class services

Services implement the behavior of components. Refactoring operations (add, remove, move), run-time behavior modification, JIT compilation and other features are possible when services do have first-class status. There are two aspects of services reflection: structural aspect and behavioral aspect. Structural reflection focus on reification of formal parts of services like name, parameters, etc. Behavioral reflection focus on reification of concepts from which behavior description is composed, *i.e.* assigments, invocations, etc. We only deal with structural reflection of services, we do not implement behavior reflection, because it may lead to inefficient programs [Malenfant *et al.*, 1992].

The reification of services is based on the analysis of services' structure. The Listing 5.9 shows an example of a service which converts miles to kilometers. It has the following structure: temporary variables (`const` and `result`), parameters (`miles`), context (`calc`, `self`, `super`) and body (code). A component diagram showing this structure is shown in Figure 5.9. Taking into account the analysis of the structure of services the `milesToKms` service from Listing 5.9 can be rewritten as shown in Listing 5.10.

Listing 5.11 shows the COMPO implementation of the `Service` descriptor. Each service has a signature (port `serviceSign` to which an instance of `ServiceSignature` descriptor will be connected), Temporary variables names and values (collection ports `tempsN[]` and `tempsV[]` ports), a program

```
service milesToKms(miles) {
  | const result |
  const := 1.609;
  result := calc.mul(miles, const);
  return result;
}
```

LISTING 5.9 : Analysis of services structure, the `milesToKms` example.

```
service milesToKms(miles) {
  tempsV[1].connectTo(1.609);
  tempsV[2].connectTo(context.getPortNamed('calc').mul(paramsV[1], tempsV[1]));
  return tempsV[2];
}
```

LISTING 5.10 : Analysis of services structure, the `milesToKms` example from Listing 5.9 in structural perspective

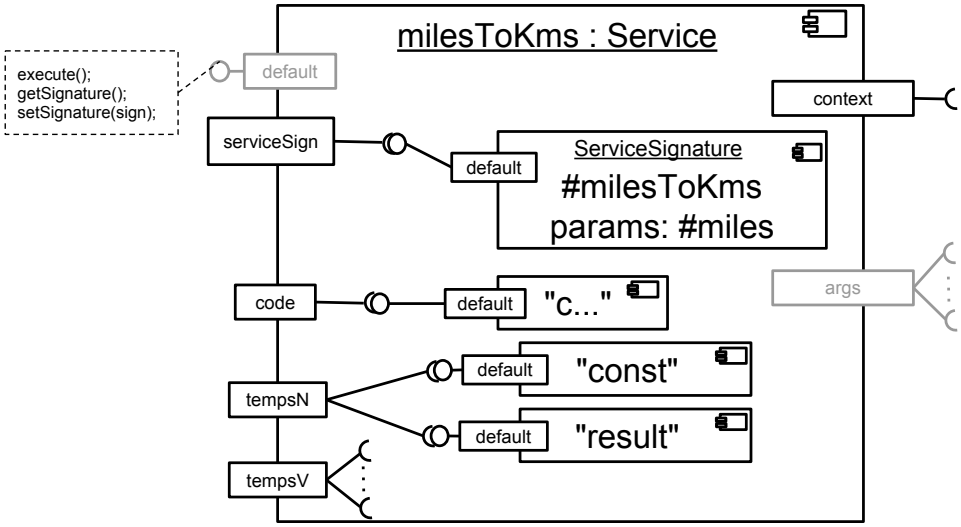


Figure 5.9 : Reification of services, the miles-to-kms example.

text (port `code`), the inherited `args[]` collection port to connect invocation arguments, an execution context (port `context`, to be connected at run-time to a component representing an execution context).

The service `execute()` checks if all requirements are satisfied (notice the use of the & operator), *i.e.* if a context component and parameter values are connected. Then it performs a system primitive to execute the code.

```
Descriptor Service extends Component
{
  provides { default : {execute(); setSignature(sign); getSinature(); }}
  requires {
    context : Component;
  }
  internally requires {
    serviceSign : ServiceSign;
    tempsN[] : Symbol;
    tempsV[] : *;
    code : String;
  }
  architecture { ... }
  service execute() {
    |bool1 bool2|
    bool1 := &context.isConnected();
    bool2 := &paramsV.isConnected();
    if( bool1 & bool2 )
    {
      return <primitive_execute>;
    }
  }
}
```

LISTING 5.11 : The SERVICE descriptor.

The SERVICESIGNATURE descriptor definition is in Listing 5.12, its instances are pretty simple, they provide services to store and access selectors, and parameter names of services.

```
Descriptor ServiceSignature extends Component {
  provides {
    default : { setSelector(name); getSelector(); getParamsCount();
                getParamName(index); setParamName(index); }
  }
  internally requires {
    name : Symbol;
    paramNames[] : Symbol;
  }...
}
```

LISTING 5.12 : The PORTDESCRIPTION descriptor.

Standard services are created with statement having the following template: `service`

`<selector>(<param1>,<param2>,...,<paramN>) { <body> }`. Having first-class services opens a way to implement *closures*-like anonymous services. Although this is an experimental feature of COMPO, it seems to be a step in a right direction. Users can create anonymous services by the statement with the following template: `[:<param1>,:<param2>,...,:<paramN>|(|<temp1>,...,<tempN>|) <code>]`, for example: `[:x, :y | return x + y; ]`.

Since services are first-class citizens it is possible to pass them as arguments of services invocations. To preserve the safety of the context component of a service (representing the environment of the service) we prevent invocations of internal services of the context component by automatic reconnection of the required port `context` of a service being passed (as argument) to the `default` port of its context component.

A (receiver) service which receives another service as an argument has to be able to connect values to the `args` port of the argument. For example, suppose that the `foreach` service has been invoked as follows: `x.foreach([:e| self.add(e)]);`, then the following code snippet shows the need for connecting the `args` port:

```
service foreach(closure) {
  |i|
  for(i:=0;i<self.size();i:=i+1)
  {
    connect args@closure to default@(self.at(i));
    closure.execute();
  }
}
```

The standard semantics of **`connect-to`** statement (see Chhoice 18) forbids to use, in the code of a service, a value of a service parameter (argument) to build a regular or delegation connection. As we explain in Section 3.3.2, such a restrictive semantics preserves the communication integrity, because it does not allow to make connections to a component (possible an internal component) which is passed as an argument. But, in case of first-class services, we believe that the potential brought by the ability to pass a component representing a service as an argument is a reasonable excuse for the following choice:

Choice 39 *As an experimental exception to the rule captured by Choice 18, we allow to use a service being passed as an argument for building connections.*

Example: invoking the `milesToKms` service In this example we show how does things go when a `map` component requires and later invokes the `milesToKms` service of a `convertor` component. Listing 5.13 shows descriptors MAP and CONVERTOR of the `map` and `convertor` components respectively. To make things more interesting, suppose that both the `map` and `convertor` components are each wrapped in a different composite, *i.e.* each is an internal component of a composite, and these composites are connected. Figures 5.10, 5.11 and 5.12 illustrate this situation and a scenario when the

service `foo` of the `map` componet invokes the `milesToKms` service of the `convertor` component. We divide the invocation scenario into 3 phases captured by the figures:

Phase 1 Figure 5.10 shows how the `foo` of `map` emits the `milesToKms` invocation through port `conv` and the way how it is passed via delegation and regular connections to the `default` port of the `convertor`. The invocation path conforms to the service invocation mechanism of COMPO (*cf.* Section 3.3.2)

Phase 2 Figure 5.11 shows how the *miles* parameter of the `milesToKms` service is automatically connected to the *trckLen* argument of the invocation. The argument passing conforms to argument passing technique of COMPO (*cf.* Section 3.3.2)

Phase 3 Figure 5.12 show how the `milesToKms` service computes and returns the result. The return value is passed back and its clone (thanks to the assignment operator `:=`) is connected to the `trckLen` internal required port of `map`. This conforms to the Choices 18 and 19 that we have made in Section 3.3.2.

```
Descriptor Map extends Component {
  requires { conv : Convertor }
  internally requires { trckLen : Number }
  service foo() {
    ...
    trckLen := 2;
    trckLen := conv.milesToKms(trckLen);
    ...
  }
}
Descriptor Convertor extends Component {
  provides { default : { milesToKms(miles); } }
  service milesToKms(miles) {
    | const result |
    const := 1.609;
    result := calc.mul(miles, const);
    return result;
  }
}
```

LISTING 5.13 : The MAP and CONVERTOR descriptors

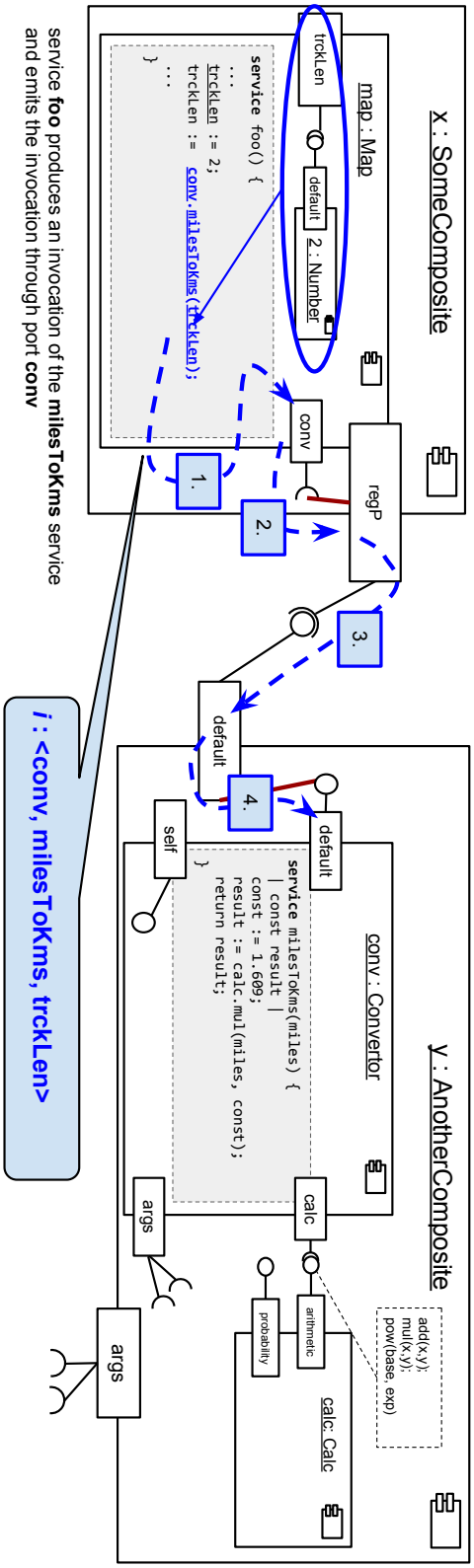


Figure 5.10 : An illustration of the `milesToKms` example 5.7 - Phase 1

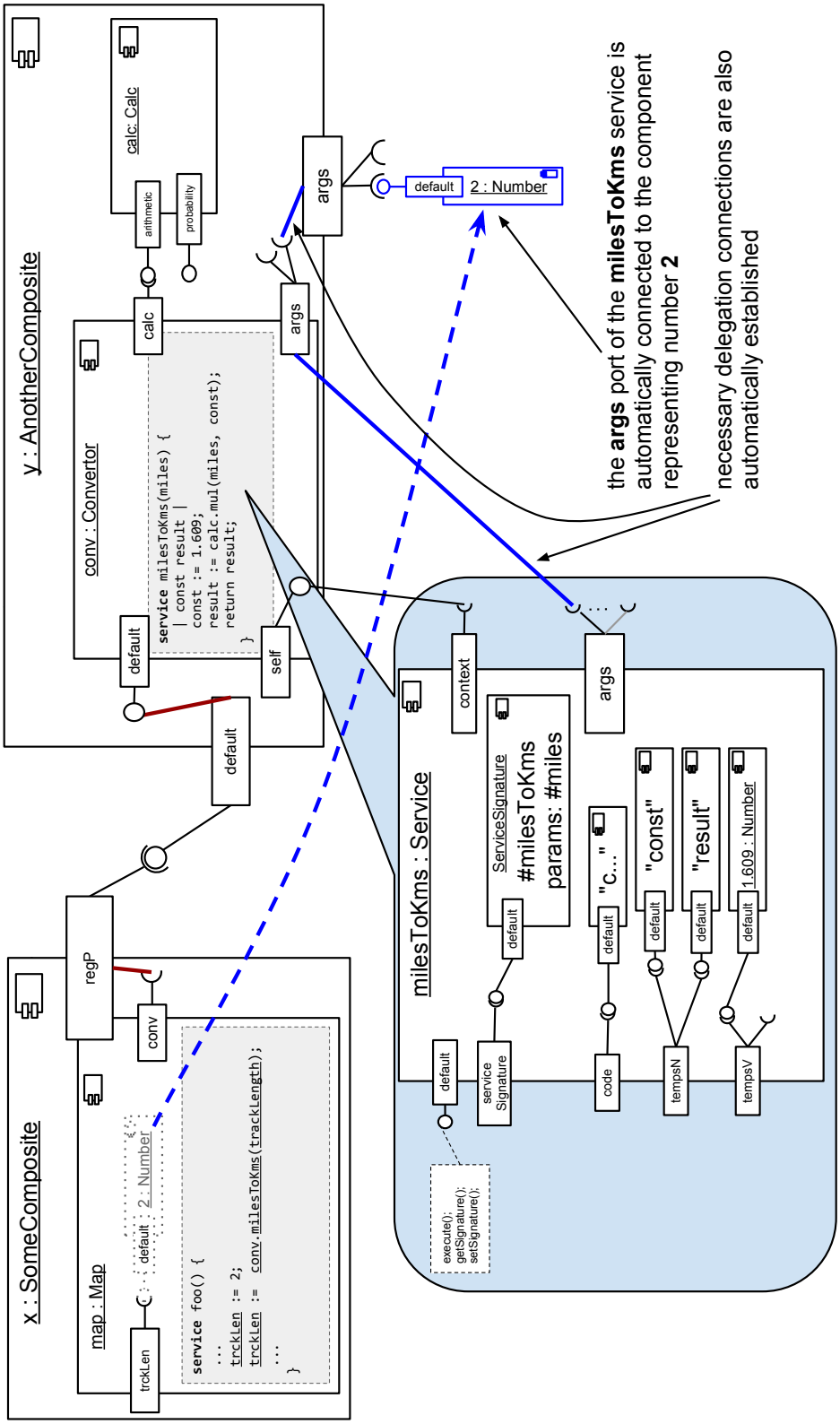
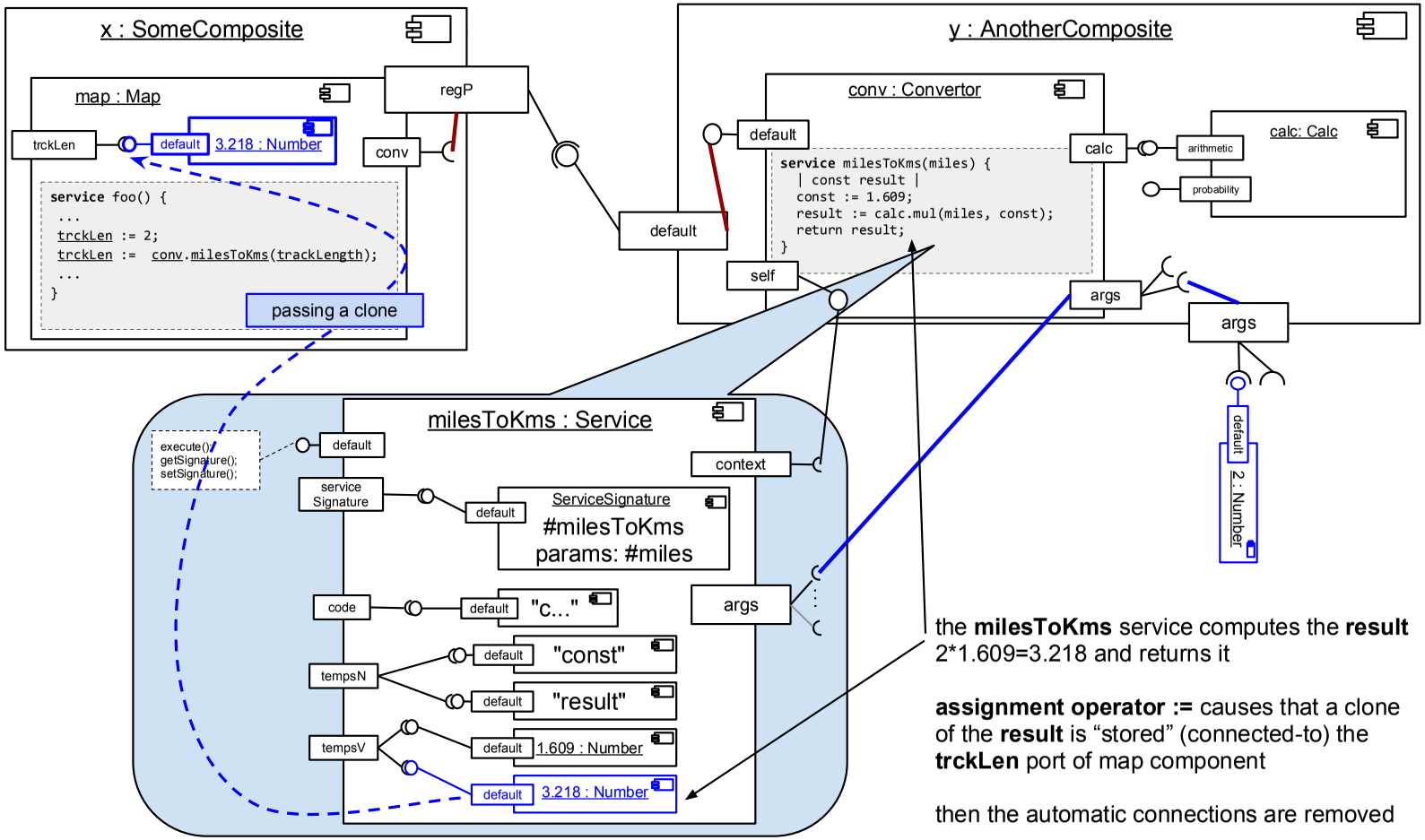


Figure 5.11 : An illustration of the `milesToKms` example 5.7 - Phase 2



the **milesToKms** service computes the **result** 2*1.609=3.218 and returns it

assignment operator := causes that a clone of the **result** is “stored” (connected-to) the **trckLen** port of map component

then the automatic connections are removed

Figure 5.12 : An illustration of the `milesToKms` example 5.7 - Phase 3

5.8 Related work

This section discuss the reflection capabilities of the three families (*cf.* Section 2.2.1) of component-based approaches.

The generative family The static nature of ADLs also do not match with reflection very well [Medvidovic et Taylor, 2000]. Reflection or at least introspection capabilities depend on code which is generated from architectures that these ADLs describe. For example, reflection is partially supported in C2 [Medvidovic *et al.*, 1996] through *context reflective interfaces*. Each C2 connector is capable of supporting arbitrary addition, removal, and re-connection of any number of C2 components. UML 2 provides support for CBSE. UML itself is not a reflective language, but its meta-model (defined with MOF [OMG, 2011a]) is. Reflection capabilities (manipulation of properties, invoke method, instance creation, etc.) provided by MOF are specifications only, i.e. there is no support for run-time reflection capabilities (as we introduced in COMPO).

The framework family Existing middleware technologies and standards provide very limited support for platform openness, usually restricted to high-level services, while the underlying platform is considered a black box. Recently, technologies such as interceptors, are a trend towards more openness. Nevertheless, the kind of openness provided is still limited to a few aspects of the platform.

CORBA Component Model (CCM) [OMG, 2012], Enterprise Java Beans (EJBs) [Oracle, 2012] or Component Object Model (COM) [Microsoft, 2012] do not provide support for explicit architecture definition, the black-box approach they support does not fit with reflection very well. Introspection interfaces, which can be used to discover the capabilities of components, are the only reflection capability they offer. For example *CCM Navigation interface* for discovering facets (provided ports) or `IUnknown` interface in COM for discovering external (client and server) interfaces of a COM object. The interface `EJBContext` defines methods to retrieve references to the bean's EJB home and remote interfaces classes, then normal Java reflection can be used to introspect the methods available to a client.

Only very few solutions consider reflection as a general approach which can be used as an overall framework that encompasses platform customization and dynamic reconfiguration. These models try to overcome the limitations of black-box approach by providing components with meta-information about their internal structure.

Projects OpenCORBA [Ledoux, 1999] and DynamicTAO [Kon *et al.*, 2000] adopt reflection as a principled way to build flexible middleware platforms.

OpenCORBA is based on the meta-class approach and on the idea of modifying the behavior of a middleware service by replacing the meta-class of the class defining that service. This is mainly used to dynamically adapt the behavior of remote invocations, by applying the above idea to the classes of stubs and skeletons.

DynamicTAO is a CORBA compliant reflective ORB, which makes explicit the architectural structure of a system in a causally connected way. Component configurators keep the consistency of dependencies as new components are added or removed from the system. Reflection capabilities are

limited to coarse-grained components, without possibility to control more detailed structures of the platform.

OpenCOM [Clarke *et al.*, 2001] (a lightweight and efficient component model based on COM) enables users to associate (dissociate) interceptor components with (from) some particular interface or to obtain all current connections between the host components' receptacles and external interfaces.

Many reflection capabilities are supported in Fractal [Bruneton *et al.*, 2006] component model, but the capabilities vary depending on kinds of *Controllers* (e.g. Attribute controller, Binding controller, Content controller, ...) a Fractal component membrane contains. The Fractal specification provides several examples of useful forms of controllers, which can be combined and extended to yield components with different introspection and intercession features. An advanced example of using controllers is FraSCAti [Seinturier *et al.*, 2012] model for development of highly configurable SCA solutions. In COMPO, reflection capabilities are the same for all components (an orthogonal model). In addition, we go further in the reification of component-level concepts: services, ports and descriptors are components.

Furthermore, middleware component models are often designed to be platform independent. Then, for each platform, the tool support of these models generate code skeletons to be filled later. Consequently run-time transformations on components and their internal structure are performed through objects and not components. For example SOFA [Hnětynka et Plášil, 2006; Bures *et al.*, 2006] reifies connectors. It is thus possible to specify high-level connectors within architecture descriptions. But finally, each primitive part of a connector specification has to be mapped by developers to some (object-oriented) code. Then reflection can be used if it is provided by this target (object-oriented) implementation language. In this case however, reflection does not address component-level concepts as in COMPO.

Models@runtime [Blair *et al.*, 2009] stream pushes the idea of reflection one step further by considering the reflection layer as a real model that can be uncoupled from the running architecture (e.g. for reasoning, validation, and simulation purposes) and later automatically re-synchronized with its running instance.

MetaORB [Costa *et al.*, 2006] proposes the design time use of models to generate MetaORB configurations, and, at run-time, the use of these same models as the causally connected self-representation of the middleware components that is maintained by the reflective meta-objects for the purposes of dynamic adaptation. MetaORB provides the meta-information management with a principled reflective meta-level. This has the benefit of unifying the use of meta-information in the system (*e.g.* preventing that different meta-object implementations use different meta-level representations), as well as providing a basis to closely integrate the configuration and adaptation features of the platform. In contrast to COMPO's orthogonal model where a change to a descriptor is propagated to all its instances, MetaORB reflection is based on per-object meta-objects, enabling to isolate the effects of reflection.

All Kevoree concepts (Component, Channel, Node, Group) obey the object type design pattern to separate deployment artifacts from running artifacts. In opposite to COMPO, where reflection capabilities are similar to all entities, Kevoree's adaptation capabilities depend on different types of nodes.

The adaptation engine relies on a model comparison between two Kevoree models to compute

a script for a safe system reconfiguration; execution of this script brings the system from its current configuration to the new selected configuration. Such adaptation scripts are written by designers, or they can be generated by automated processes (*e.g.* within a control loop managing the Kevoree system). In fact, the adaptation scripts are comparable to model transformations written in COMPO.

The above described component models provide many sophisticated means for creating adaptable dynamic component-oriented solutions, but, in opposite to component-based programming languages like COMPO, they use object-oriented programming to implement component-based software. Therefore there is no continuum to achieve the various stages of component-based software development using the same conceptual model.

The Component-oriented languages family The big advantage of component-oriented languages (COLs) is that they do not separate architectures from implementation and so they have potential to manipulate reified concepts. In opposite to COMPO, component-level concepts are often reified as objects, instead of components. This leads to a mixed use of component and object concepts. For example reflection package of ArchJava [Aldrich *et al.*, 2002] specifies class (not component class) `Port` which represents a port instance. Very often the representations are not causally connected to concepts they represent. In case of ArchJava, which relies on Java reflection, the reason is that reflection in Java is mostly read-only, i.e. introspection support only.

Reflection is not explicitly advocated in ComponentJ [Seco *et al.*, 2008]. It however appears that a running system certainly has a partial representation of itself to allow for dynamic reconfiguration of internal architectures of components as described in [Seco *et al.*, 2008] but it seems to be a localized and ad.hoc capability, the reification process being neither explicited nor generalized as in our proposal.

5.9 Summary

In this chapter, we have described an original meta-model for a reflective component-based programming language allowing for standard application development, and for static or run-time model and program transformations. We have proposed concrete, adapted (first-class descriptors) or new (first-class ports), dmd meta-level solutions for a component-based reification of concepts leading to a "*everything is a component*" operational development paradigm.

Such a reflective language offers a continuum to achieve the various stages of component-based software development in the context of one language. For example a programmer can design a component-oriented architecture, then verify the architecture's properties and then seamlessly fill it in with code, all using COMPO. As a reflective language gives access (via meta-components) to elements of the component-based meta-model, COMPO also makes it possible to design and implement new component-based constructs (as exemplified with achieving a new kind of ports).

A key issue is uniformity. We have described a full component-based meta-model and a reflective description in COMPO of its main component descriptors made executable via a concrete implementation. This opens the essential possibility that architectures, implementations and transformations can all be written at the component level and using a unique language. The final solution is thus extensible and permits to achieve various applications and modeling scenarios.

The following sub-section is a recapitulation of the definitions and the choices that we have made for COMPO in this chapter.

5.9.1 Definitions made

Primitive port A primitive port is a rock-bottom entity that cannot be created by users and cannot be used as a first-class entity. It implements the behavior of standard ports. Every port declared within the PORT descriptor is automatically `primitive`. (*cf.* 20

The & operator The & operator applied on a port, *i.e.* `&<portName>`, returns on demand created primitive internal required port, which is automatically connected to the `default` port of the component representing (reifying) port `<portName>`. Is true that `&<portName> == &&<portName>` (double application returns the same result). (*cf.* 21)

5.9.2 Choices made

Choice 34 Descriptors, ports and services are true components, instances of descriptors DESCRIPTOR, PORT and SERVICE respectively.

Choice 35 Descriptor COMPONENT is a basic descriptor and the root of inheritance tree, all descriptors inherit it

Choice 36 Descriptor DESCRIPTOR is a sub-descriptor of descriptor COMPONENT. It describes descriptors (it is a meta-descriptor) and is the instance of itself.

Choice 38 A port is a component having primitive ports.

Choice 37 A port declaration can be made more specific by putting the following statement **`ofKind`** `<descriptor>` after interface specification. The statement specifies that a port will be created as an instance of the specified descriptor `<descriptor>`.

Choice 39 As an experimental exception to the rule captured by Choice 18, we allow to use a service being passed as an argument for building connections.

CHAPTER 6

COMPO in Practice

If A is success in life, then A = x + y + z. Work is x; y is play; and z is keeping your mouth shut.

Albert EINSTEIN.

Preamble

Smaller examples of using composition, inheritance, introspection, intercession and meta-modeling have already been given in the previous chapters. In this chapter we present a complete architecture design of the HTTP server which was used previously in these examples. Then we present an example of hierarchy modeling, where COMPO*'s inheritance system plays the main role. We also benchmark* COMPO*'s reflection capabilities and its meta-model in an example of architecture transformation and architecture constraint verification. We show how architecture constraints can be executable and reusable.*

6.1 Designing an HTTP server

IN this example we present a descriptor named HTTPSERVER (*cf.* Listing 6.1) which models a simple HTTP server that receives HTTP requests from network, processes these requests and finally creates and sends the responds. The intent of this example is to show the architecture description power of COMPO.

The descriptor provides the services `run` and `status` through the `default` provided port. It states that a server is composed of two internal components, an instance of FRONTEND accessible via the internal required port `fE`, and an instance of BACKEND accessible via the internal required port `bE`. These internal components are connected together so that the `front-end` can invoke services of the `back-end`. The HTTPSERVER descriptor explicitly defines the implementation of the `status` service. The provided service `run` is implemented by a delegation connection to the provided port `default` of the `front-end`. Figure 6.1 shows a diagram that represents such a server component.

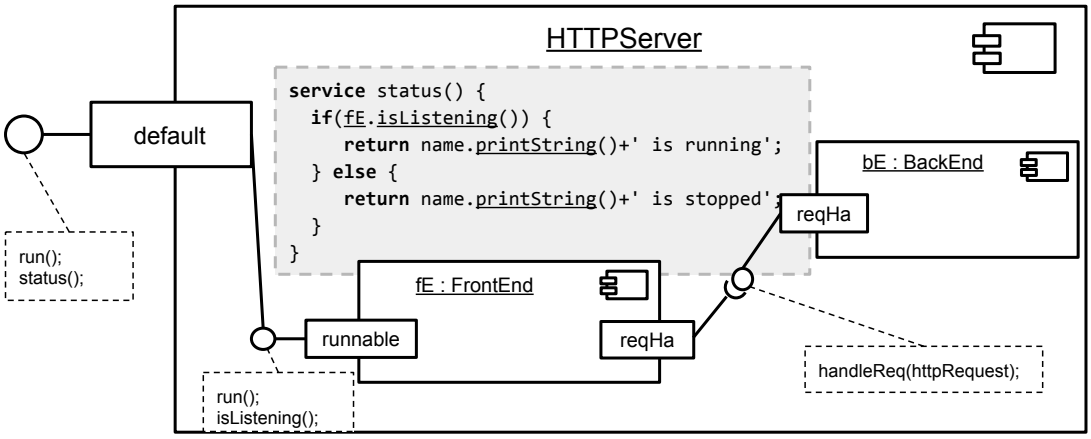


Figure 6.1 : The diagram shows a logical representation of an instance of the HTTPSERVER descriptor presented in Listing 3.1, after it has been created and initialized.

Listing 6.2 shows the FRONTEND descriptor. The descriptor specifies one provided port named `runnable` providing the services `run` and `isListening` service. It also specifies one required port named `backEnd` through which it requires the `handleReq(r)` service. Internally, it defines two internal required ports. The first port is named `rR` (request receiver) and described by the interface of the default provided port (*see* Section 10) of the REQUESTRECEIVER descriptor. The second port is named `s` and described by the interface of the default provided port of the TASKSCHEDULER descriptor. In the architecture section, we declare two delegation connections and three regular connections (*see* Section 3.2.4 for more details about delegation and regular connections.) The first delegation connection says that the external provided port `runnable` delegates service invocations to the `default` port of the internal component connected to the `rR` internal required port. The second delegation connection says that the external required port `handler` of the internal component connected to the `rR` internal required port delegates service invocations to the `backEnd` required port. The first two regular connections connect new instances of TASKSCHEDULER and REQUESTRECEIVER descriptors to the internal

```
Descriptor HTTPServer {
  provides {
    default : { run(); status() }
  }
  internally requires {
    fE : FrontEnd;
    bE : BackEnd;
  }
  architecture {
    connect fE to default@(FrontEnd.new());
    connect bE to default@(BackEnd.new());
    delegate default@self to default@fE;
    connect backEnd@fE to default@bE;
  }
  service status() {
    if(fE.isListening())
    {
      return name.printString()  + ' is running'
    }
    else
    {
      return name.printString() + ' is stopped'
    }
  }
}
```

LISTING 6.1 : The `HTTPServer` descriptor.

required ports `s` and rR respectively. The last regular connection defines that the two internal components (request-receiver and task-scheduler) are interconnected. Finally the descriptor implements services `isListening` and `run`. Diagram in Figure 6.2 shows an instance of the FRONTEND descriptor.

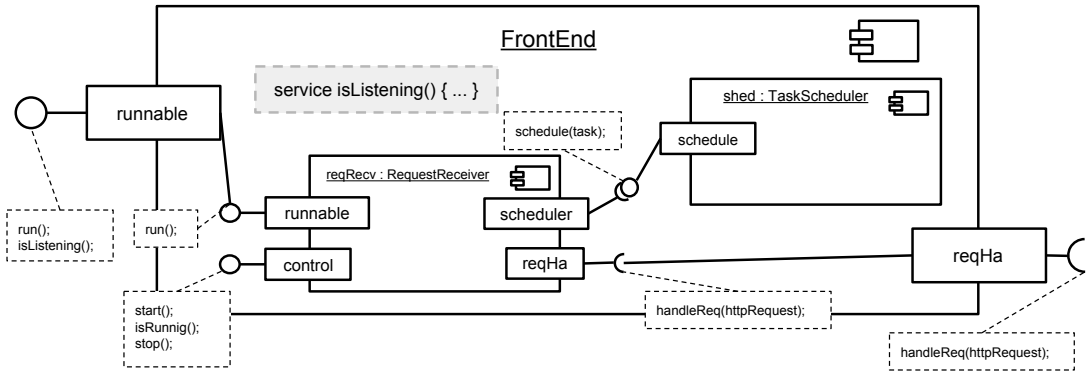


Figure 6.2 : A diagram of an instance of the FRONTEND descriptor

The `back-end` component processes and creates responds for the requests. For each request, the BACKEND descriptor creates an instances of REQUESTHANDLER descriptor. Listing 3.2 shows the BACK-

```
Descriptor FrontEnd {
   provides {
      runnable : { run(); isListening(); }
   }
   requires {
      backEnd : { handleRequest(r) }
   }
   internally requires {
      rR : RequestReceiver;
      s : TaskScheduler;
   }
   architecture {
      delegate default@self to default@rR;
      delegate handler@rR to backEnd@self;
      connect s@self to default@(TaskScheduler.new());
      connect rR@self to default@(RequestReceiver.new());
      connect rR@self to schedule@self;
   }
   service isListening() {
       return rR.isRunning();
    }
    service run() {
       ...
    }
}
```

LISTING 6.2 : The `FrontEnd` descriptor

END descriptor. The descriptor specifies one provided port named `reqHa` (request handler), providing the `handleReq(r)` service.

Internally, it defines three internal required ports. The first port is named `analyzer` and described by the interface of the default provided port (*see* Section 10) of the REQUESTANALYZER. The second port is named `logger` and described by the interface of the default provided port of the LOGGER. The third port is a collection port named `handler` and described by the interface of the default provided port of the REQUESTHANDLER.

In the architecture section, we declare one delegation connection and three regular connections (*see* Section 3.2.4 for more details about delegation and regular connections.) The delegation connection says that the external provided port `reqHa` delegates service invocations to the `inReqHa` port of the internal component connected to the `analyzer` internal required port. The first two regular connections connect new instances of LOGGER and REQUESTANALYZER descriptors to the internal required ports `logger` and analyzer respectively. The last regular connection defines that the two internal components are interconnected between the `logger` and `logging` ports. Finally we can see the implementation of the service `addHandler` service which dynamically adds and connects new instances of the REQUESTHANDLER descriptor. The newly created components are connected to the `handlers` internal required collection port and then each newly created component is connected to the `outReqHa` external required collection port of the `analyzer` component, which is an instance of

```
Descriptor BackEnd {
  provides {
    reqHa : { handleReq(httpRequest); }
  }
  requires {
    reqHa : { handleReq(r) }
  }
  internally requires {
    analyzer : RequestAnalyzer;
    logger : Logger;
    handlers[] : RequestHandler
  }
  architecture {
    delegate reqHa to inReqHa@analyzer;
    connect logger to default@(Logger.new());
    connect analyzer to default@(RequestAnalyzer.new());
    connect logger@analyzer to logging@logger;
  }
  service addHandler()  {
    |i|
    i := connect handlers to default@(RequestHandler.new());
    connect outReqHa@analyzer to reqHa@handlers[i];
  }
}
```

LISTING 6.3 : The `BackEnd` descriptor

the REQUESTANALYZER shown in Listing 6.4. Diagram in Figure 6.3 shows an instance of the BACKEND descriptor.

```
Descriptor RequestAnalyzer {
  provides {
   inReqHa : { handleReq(req, index); }
  }
  requires {
    logger : { log(str) };
   outReqHa[] : { handleReq(httpRequest); };
  }
  ...
}
```

LISTING 6.4 : The `RequestAnalyzer` descriptor

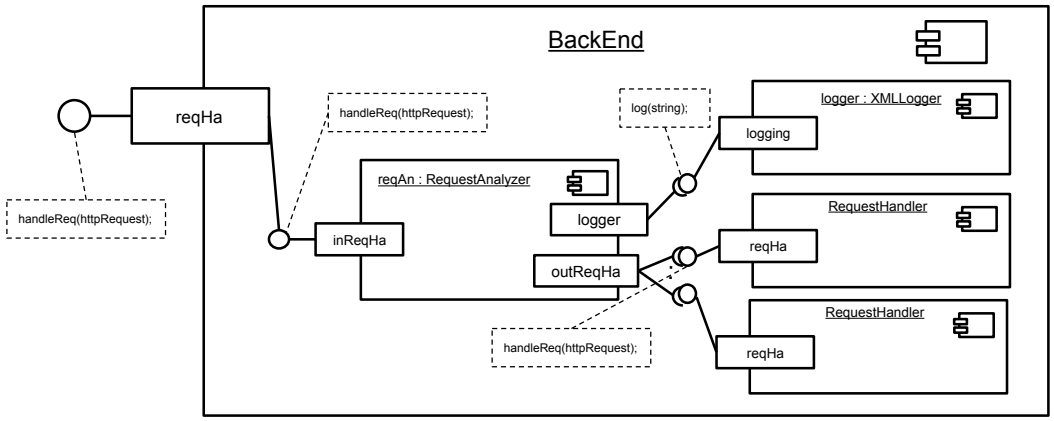


Figure 6.3 : A diagram of an instance of the BACKEND descriptor

6.2 Designing a collection hierarchy

The collection hierarchy example was inspired by group of classes that appears in the "*Blue Book*" of SMALLTALK [Goldberg et Robson, 1989]. The group contains 17 sub-classes of Collection and has already been redesigned several times before the SMALLTALK-80 system was released. This group of classes is often considered to be a paradigmatic example of object-oriented design.

In the following, we redesign a part of the collections hierarchy in the component-based context to present modeling capabilities provided by COMPO. We use COMPO inheritance mechanism and abstract descriptors to build a three-level hierarchy having the following structure:

```
Collection                      /* level 0 */
|-- AbstractSet                  /* level 1 */
|  |-- IdentitySet               /* level 2 */
|  |-- Set                       /* level 2 */
|
|-- Bag                          /* level 1 */
|-- SequenceableCollection       /* level 1 */
   |-- Stack                     /* level 2 */
```

We design the hierarchy starting from the level zero, *i.e.* we start with the root component described by abstract descriptor COLLECTION, after that we will design the levels one and two. The main purpose of this descriptor is to define the basic external contract common to all collections. The descriptor defines that every collection will have 3 ports, each for a different protocol, the ports are named: `accessing`, `testing` and `enumerating`. Even if the descriptor is abstract, some services like `addAll(coll)` or `isEmpty()` can be implemented. The `select` service evaluates *closure* with each of

the receiver's elements as the argument and collects into a new collection like the receiver, only those elements for which *closure* evaluates to true. We show the COLLECTION descriptor in Listing 6.5.

```
Descriptor Collection extends Component
{
   provides {
      accessing : { size(); add(item); addAll(coll); remove(item); removeAll(); };
      testing : { isEmpty(); };
      enumerating : { select(closure); foreachDo(closure)};
   }
   service isEmpty() {
      return self.size() == 0;
   }
   service addAll(coll) {
      coll.foreachDo([:each | self.add(each); ]);
   }
   service removeAll() {
      self.foreachDo([:each | self.remove(each); ]);
   }
   service select(closure) {
      |collCopy|
      collCopy := self;
      coll.removeAll();
      self.foreachDo([:each|
         connect args@closure to default@each;
         if(closure.execute())
         { collCopy.add(each); }
      ]);
      return collCopy;
   }
}
```

LISTING 6.5 : The COLLECTION abstract descriptor

The next step is to create level one descriptors:

ABSTRACTSET (*cf.* Listing B.1 in Appendix B.1): represents an abstract descriptor of all sets. It declares or implements services which are common for all sets. The descriptor implements the `foreachDo` service and the not public service `indexOf(item)`. It also declares new not public services `getItemAt(index)` and `areEqual(itemA, itemB)` which its sub-descriptors should implement. The not public services are not externally provided because users of sets should not be able to access items by indexes.

BAG (*cf.* Listing B.2 in Appendix B.1): representing an unordered collection of possibly duplicate elements.The BAG descriptor is very similar to the SET descriptor, but it allows to store the same item multiple times. For that purposes it tracks (using internal required collection port `tally`) the count of each item in the bag.

SEQUENCEABLECOLLECTION (*cf.* Listing B.3 in Appendix B.1): representing collections that have a well-defined order associated with their elements. Thus each element is externally-named by inte-

gers referred to as indices. The descriptor specializes interfaces of ports `accessing`, `removing` and `testing` to provide new services related with mapping elements to indices.

Finally we implement the second level children, *i.e.* a sub-descriptor of SET and a sub-descriptor of SEQUENCEABLECOLLECTION:

STACK (*cf.* Listing B.4 in Appendix B.1): representing stacks. The STACK descriptor extends SEQUENCEABLECOLLECTION descriptor, with a new provided port `stackable` through which it provides services `push` and `pop`. It specializes services of SEQUENCEABLECOLLECTION in a way that they throw an error if the items of a stack are manipulated directly with an index.

SET (*cf.* Listing B.5 in Appendix B.1): representing a set of components without duplicates. The descriptor has an additional internal required collection port `items` in order to store items and manage duplicates. In opposite to the IDENTITYSET descriptor (described below), a set stores clones (copies with a new identity) of items and therefore it is possible to connect the items to the internal required port and thus encapsulate the items within the set. The cloning ensures communication integrity, for example, an internal component of a composite can be stored into a set without breaking the communication integrity because it is not possible to invoke services of the internal component of the composite from the set (since the set contains clones).

IDENTITYSET (*cf.* Listing B.6 in Appendix B.1): representing the same as a Set, except that the stored items are not clones of real components. To be able to store components to which a third party component might be connected, we have to introduce a new external required port `items`, in opposite to internal required port `items` of SET. Because `items` port is external, it would be possible to break the integrity of sets by, for example, connecting items directly to the port, *i.e.* not via service `add`. To solve this issue, we developed a new kind of collection port (descriptor SETCOLLPORT, *cf.* Listing B.7), which specializes the `connectTo` and `disconnect` services to keep the integrity of sets. These services are modified in a way they verify the identity of a newly connected item (`connectTo`) and update the size of a set when items are connected and disconnected. The `items` port is instance of SETCOLLPORT and thus it makes it possible to store items externally and in the same time to preserve the integrity of identity sets.

6.3 Transformation to a bus-oriented architecture

Model transformations are a key issue for MDE [Carrière *et al.*, 1999]. A wide range of model transformation languages and tools exist. While transformation experts need to understand the transformation language and the source and target domains, domain engineers understand the source and target domains/languages but have no skills in the model transformation language. By putting reflection into COMPO we have opened the possibility to write various kinds of model or program transformations and verifications

The transformation scenario performed on COMPO's implementation of the simple HTTP server, described in Section 3.2.1 migrates its component-based architecture from a classic front-end/back-end architecture into a bus-oriented architecture. The transformation (sketched in Figure 6.4) is motivated by a use-case when a customer (already running the server) needs to turn the server with

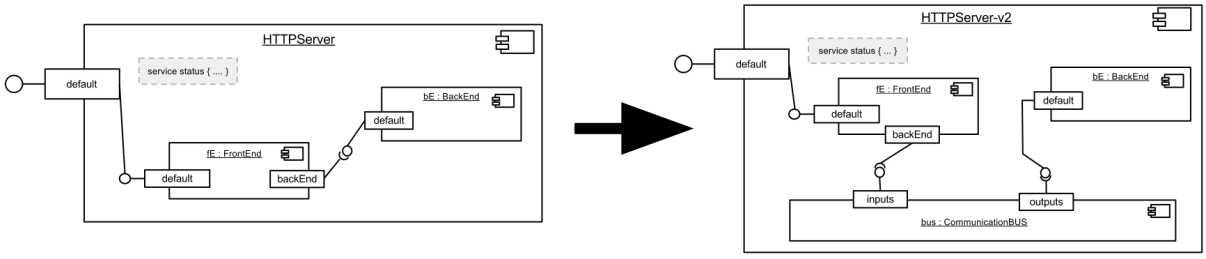


Figure 6.4 : Simplified diagram illustrating the transformation from a classic front-end back-end architecture into a bus-oriented one.

multiple fronts-ends and back-ends into a bus-oriented architecture which reduces the number of point-to-point connections between communicating components. This, in turn, makes impact analysis for major software changes simpler and more straightforward. The tax to pay for the increased flexibility coming with bus-oriented architecture is an increased overhead and a slower communication speed.

The transformation is modeled with a descriptor named TOBUSTRANSFORMER, *see* Listing 6.6. An instance of the TOBUSTRANSFORMER descriptor can be connected to the HTTPSERVER descriptor (seen as a component) (COMPO's code in Listing 3.1) and the following transformation steps could be performed:

Step 1 introduce a new internal required port named `bus` to which an instance of a `Bus` descriptor (not specified here) will be connected;

Step 2 remove the original connection from front-end to back-end.

Step 3 extend the original architecture with new connections from front-end and back-end to bus;

The following code snippet shows the use of the transformation component. Let us suppose that `server` represents a port connected to an instance of the HTTPSERVER descriptor

```
transformer := ToBusTransformer.new();
connect target@transformer to default@HTTPServer;
transformer.transform();
```

```
Descriptor ToBusTransformer {
 requires { target : IDescriptor }

 service step1-AddBus() {
  |pd cd|

  pd := PortDescription.new();
  pd.setName('bus');
  pd.setRole('required');
  pd.setVisibility('internal');
  pd.setInterface('IBus);

  target.addPortDescription(pd);

  cd := ConnectionDescription.new();
  cd.setSourcePort('bus');             /* bus@self */
  cd.setSourceComponent('self');
  cd.setDestinationPort('default');    /* default@(bus.new()) */
  cd.setDestinationComponent('Bus.new()');

  target.addConnectionDescription(cd);
 }

 service step2-RemOldConns() {
  |cd|
  cd := DisconnectionDescription.new();

  cd.setSourcePort('backEnd');         /* backEnd@fE */
  cd.setSourceComponent('fE');
  cd.setDestinationPort('default');    /* default@bE */
  cd.setDestinationComponent('bE');

  context.removeConnectionDescription(cd);
 }

 service step3-ConnectAllToBus() {
  |cd|
  cd := ConnectionDescription.new();
  cd.setSourcePort('backEnd');        /* backEnd@fE */
  cd.setSourceComponent('fE');
  cd.setDestinationPort('inputs');    /* inputs@bus */
  cd.setDestinationComponent('bus');

  target.addConnectionDescription(cd);

  cd := ConnectionDescription.new();
  cd.setSourcePort('default');         /* default@bE */
  cd.setSourceComponent('bE');
  cd.setDestinationPort('outputs');    /* outputs@bus */
  cd.setDestinationComponent('bus');

  target.addConnectionDescription(cd);
 }
}
```

LISTING 6.6 : The `ToBusTransformer` descriptor.

6.4 Verifying architecture constraints

A part of architecture decision documentation [Tang *et al.*, 2005 ; Kruchten *et al.*, 2009] is composed of architecture constraints. Examples of constraints include the verification of a particular architectural style or pattern, like the layered style. This kind of documentation often includes some parts which can be used individually for documenting parts of design decisions [Tibermacine *et al.*, 2010b]. Unfortunately, there is no mean to specify these parts and to make them parametrized entities that can be factorized and used in different reuse contexts.

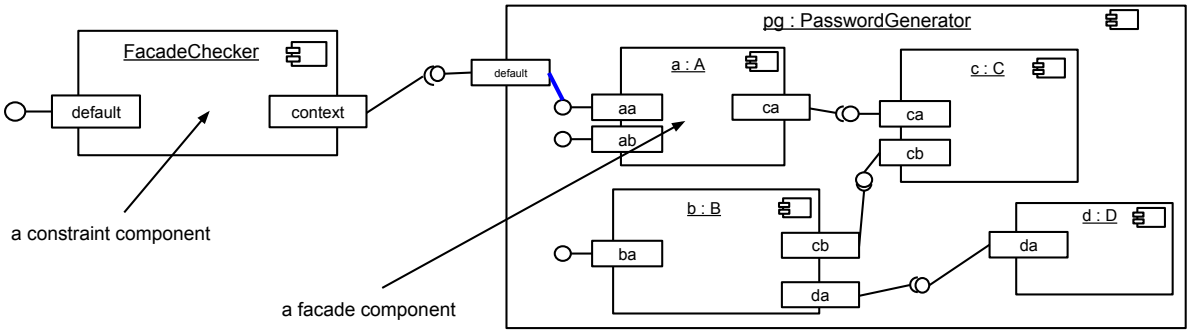


Figure 6.5 : The facade checker constraint component is connected to an instance of PASSWORDGENERATOR descriptor in order to verify the constraint.

In our previous work, we have also studied the idea of defining blocks of constraints as customizable and reusable entities [Tibermacine *et al.*, 2010a]. And we have proposed a way to build basic constraints as checkable entities embedded in a special kind of software components, which can be reused, assembled, composed into higher-level ones and customized using standard component-based techniques. The purpose is to put reusable constraint-component on shelves and as well as to produce new constraints by composition of existing ones [Tibermacine *et al.*, 2011].

```
Descriptor Constraint extends Component
{
    provides { default : {verify();} }
    requires { context : * }
}
```

LISTING 6.7 : The CONSTRAINT descriptor

COMPO, thanks to the reflection, directly supports this idea by providing a uniform paradigm to develop business and non-functional (constraint-) components. COMPO users may build constraint components by creating sub-descriptors of the CONSTRAINT descriptor (*see* Listing 6.7), which is the basic abstract descriptor for all constraint descriptors.

Every constraint must be assigned to an unambiguous context that defines the target for references within the constraint. In COMPO, business components represent a context for constraint components. The CONSTRAINT descriptor defines a required port `context` to which the users may connect

their business components, as it is shown in Figure 6.5. The interface of the `context` port might be specialized to specify the context more precisely.

By default, every constraint descriptor should provide a boolean service called `verify` to verify its current context.

The following examples show: (i) a simple constraint component (VERIFYBUSARCH descriptor) and; (ii) a composite constraint component (PIPEANDFILTER descriptor) built from reusable components.

6.4.1 Verifying the bus-oriented architecture

In this example, we design a constraint descriptor VERIFYBUSARCH to verify if the result of the "*to bus transformation*" example (*cf.* Section 6.3) conforms to a bus-oriented architecture. We will connect an instance of the VERIFYBUSARCH descriptor, *i.e.* a contraint component, to an instance of the HTTPSERVER (*cf.* Listing 3.1), *i.e.* a business component, in order to perform post-transformation verification. The constraint component executes a service `verify` which does the following steps:

Step 1 verifies the presence of an internal component representing a bus, *i.e.* an instance of descriptor BUS;

Step 2 verifies that the bus component has one input and one output port;

Step 3 verifies that all the other components are connected to the bus only and the original delegation connection is preserved.

We show the COMPO's code of the VERIFYBUSARCH descriptor in Listing 6.8. The descriptor defines the `verify` service that gradually triggers services implementing the above described steps, *i.e.* services: `stepOne-IsBusPresent`, `stepTwo-HasBusIOPorts` and `stepThree-AreAllConnsToBus`. The `verify` service collects boolean results of the "step"-services and return true if all the results are true.

The following code snippet shows the use of the transformation and verification components: let us suppose that `server` represents a port connected to an instance of the HTTPSERVER descriptor

```
transformer := ToBusTransformer.new();
constraint := VerifyBusArch.new();

connect context@transformer to default@HTTPServer;
connect context@constraint to default@HTTPServer;

transformer.transform();
constraint.verify();
```

```
Descriptor VerifyBusArch extends Constraint
{
 service verify() {...}
 service stepOne-IsBusPresent() {
  |ports|
  ports := context.getDescribedPorts();
  if(ports.select([:p|
        p.getInterface()==IBus])
        .size()==1 )
  { return ports.select(
            [:p|p.getInterface()==IBus])}
  else { return false };
 }
 service stepTwo-HasBusIOPorts(busPD){
  |ports|
  ports := Bus.getDescribedPorts();

  if(ports.any([:p|p.getName()=='input']))
  { return true } else { return false };

  if(ports.any([:p|p.getName()=='output']))
  { return true }  else { return false };
 }
 service stepThree-AreAllConnsToBus(busPD){
  |conns|
  conns := context.getConnsDescs();
  conns.remove([:cd|cd.getSrcPort()
                 .getInterface()==IBus]);

  if((conns.remove([:cd|
      cd.isDelegation()]))
  {} else { return false };

  if(conns.forEach([:cd|
   (cd.srcPortDesc()==busPD)
   .or([cd.destPortDesc()==busPD])
  ]) {return true } else { return false };
 }
}
```

LISTING 6.8 : The `VerifyBusArch` descriptor.

6.4.2 Verifying the Pipe & Filter architecture

In this example we show a composite constraint component described by descriptor PIPEANDFILTER (shown in Listing 6.9) which verifies whenever or not the internal architecture of a connected business component (representing a context for the constraint) conforms to the Pipes&Filters architecture.

> "The *Pipes and Filters* architectural pattern provides a structure for systems that process a stream of data. Each processing step is encapsulated in a filter component. Data are passed through pipes between adjacent filters. Recombining filters allows you to build families of related filters." [Buschmann *et al.*, 2008]

The Pipe &Filter architecture consists of a chain of processes or other data processing entities (like components), arranged so that the output of each element of the chain is the input of the next. They are most efficiently implemented in a multitasking operating system, by launching all processes at the same time, and automatically servicing the data read requests by each process with the data written by the upstream process. In this way, the CPU will be naturally allocated alternatively among the processes by the scheduler so as to minimize its idle time. Process pipelines were invented by Douglas Mcllroy [McIlroy, 1968 ; McIlroy, 1972], one of the designers of the first Unix shells, and greatly contributed to the popularity of that operating system. It can be considered the first non-trivial instance of software components.

The PIPEANDFILTER descriptor is composed of 5 reusable constraint components, in the following we refer to these as sub-constraints. For example, the first sub-constraint can be reused as a part of "facade" constraint which checks whenever a descriptor describes the facade architecture[1].

SUBCONSTRAINTONE : There is only one internal component having one or more external provided ports connected uniquely to the owning composite only (delegated ports). The component have to declare one or more external required ports, each being connected to an internal component of the same hierarchical level, or being not connected at all. See Listing 6.10.

SUBCONSTRAINTTWO : There is only one internal component having one or more external required ports connected uniquely to the owning composite only (delegated ports). The component have to declare one or more external provided ports, each being connected to an internal component of the same hierarchical level, or being not connected at all. See Listing B.8 in Appendix B.2.

SUBCONSTRAINTTHREE : Other internal components (n-2) have external provided and required ports connected to other internal components of the same hierarchical level. See Listing B.9 in Appendix B.2.

SUBCONSTRAINTFOUR : Connection between each pair of internal components should go in the same direction, i.e. there are not two connections of opposite direction between each pair. See Listing B.10 in Appendix B.2.

[1] The facade architecture is a component-based realization of the facade pattern, *cf.* http://en.wikipedia.org/wiki/Facade_pattern.

```
Descriptor PipeAndFilter extends Constraint
{
    internally requires {
        scOne : SubConstraintOne;
        scTwo : SubConstraintTwo;
        scThree : SubConstraintThree;
        scFour : SubConstraintFour;
        scFive : SubConstraintFive;
    }
    architecture {
      connect scOne to default@(SubConstraintOne.new());
      connect scTwo to default@(SubConstraintTwo.new());
      connect scThree to default@(SubConstraintThree.new());
      connect scFour to default@(SubConstraintFour.new());
      connect scFive to default@(SubConstraintFive.new());

        delegate context@scOne to context@self;
        delegate context@scTwo to context@self;
        delegate context@scThree to context@self;
        delegate context@scFour to context@self;
        delegate context@scFive to context@self;
    }
    service verify() {
        |c1 c2 c3 c4 c5 |
        c1 := scOne.verify();
        c2 := scTwo.verify();
        c3 := scThree.verify();
        c4 := scFour.verify();
        c5 := scFive.verify();

        return (c1 & c2 & c3 & c4 & c5);
    }
}
```

LISTING 6.9 : PipeAndFilter constraint in COMPO

SUBCONSTRAINTFIVE : For each pair (A, B) of directly connected internal components, there is not a third component, which is connected to the required ports of A and in the same time to provided ports of B. See Listing B.11 in Appendix B.2.

```
Descriptor SubConstraintOne extends Constraint
{
    service verify() {
        |retval|
        retval := true;
        intComps := context.getPorts().select([:p |
            &p.isRequired().and([&p.isInternal()]);
        ]);
        intComps.each([:ic |
            ic.getPorts().each([:x |
                if(&x.isProvided().and([&p.isExternal()])) {
                    | count |
                    &x.getConnectedPorts().each([:cp |
                        if(&cp.isProvided().and([&p.isExternal()])) {
                            if(&cp.getOwner() == context.yourself())
                            { retVal := retVal.and([true]); }
                            else
                            { retVal := retVal.and([false]); }
                        }
                    ]);
                }
                if(&x.isRequired().and([&p.isExternal()])) {
                    &x.getConnectedPorts().each([:cp |
                        if(&cp.isProvided().and([&p.isExternal()])) {
                            if(&cp.getOwner().getOwner() == context.yourself())
                            { retVal := retVal.and([true]); }
                            else
                            { retVal := retVal.and([false]); }
                        }
                    ]);
                }
            ]);
         ]);
        return retVal;
    }
}
```

LISTING 6.10 : PipeAndFilter, the sub constraint one in COMPO. There is only one internal component having one or more external provided ports connected uniquely to the owning composite only (delegated ports). The component have to declare one or more external required ports, each being connected to an internal component of the same hierarchical level, or being not connected at all.

6.5 Summary

The intent of this chapter was to show COMPO's aim to provide a continuum to achieve the various stages of component-based software development in the same conceptual world. We have presented an example of an HTTP server showing architecture modeling skills and an example of hierarchy modeling which benchmarks COMPO's inheritance system. Then we have shown an example of architecture transformation and architecture constraint verification made within COMPO. This was possible thanks to the reflection capabilities which, for example, makes it possible to create executable and reusable architecture constraints.

The examples presented show that the continuum opens the essential possibility that architectures (plus their implementation), transformations and verifications can all be written at the component level and using a unique language.

CHAPTER 7

The prototype

Make it work.
Make it work right.
Make it work right and fast.

Edsger DIJKSTRA, Donald KNUTH, C.A.R. HOARE

Preamble

This chapter presents the prototype of COMPO. *The prototype is implemented in* SMALLTALK *and more precisely Pharo, which is an implementation of* SMALLTALK. *We present the prototype, technology choices, its architecture and the final implementation. We also present a draft* SMALLTALK *development environment for* COMPO.

7.1 Why Smalltalk?

Although COMPO can be implemented in different languages, we have chosen SMALLTALK, because its meta-model is extensible enough to support another meta-class system as shown in [Briot et Cointe, 1989 ; Ducasse et Gîrba, 2006]. Concretely, the prototype of COMPO was made in Pharo [Black *et al.*, 2009][1] which is a free, modern and portable implementation of SMALLTALK-80 [Goldberg et Robson, 1989].

SMALLTALK is a reflective and dynamically typed object-oriented language. Dynamically typed languages offer flexibility and qualities yet recognized [Nierstrasz *et al.*, 2005]. The main features of the SMALLTALK object model is single inheritance and the notion of implicit meta-class that is to say, a meta-class is automatically created for each user-defined class without the programmer intervention. Because of reflection, SMALLTALK is a uniform programming language ("everything is an object") and opened because it is easy to write its extensions. Pharo is an implementation of SMALLTALK written in SMALLTALK[2] therefore it is easy to write Virtual Machine extensions. Moreover, Pharo is a development environment containing many tools like the class browser, the debugger or the profiler which make the development of prototypes faster and easier.

We also found that the majority of current components-oriented languages prototypes are extensions of Java. In opposite to Java, SMALLTALK is dynamically typed and therefore seems worthwhile to offer an alternative in a different environment to better distinguish the specificities of component-oriented languages, from those of the Java language.

7.2 Technology choices

The implementation of a programming language requires the development of a chain starting form source code analysis (*parser*) and abstract syntax tree (AST) building to machine code generation or AST-interpretation. Of course, there are many tools to facilitate the construction of this chain as for example the *compiler compiler*[3] that can generate source code of a *parser*, interpreter or compiler from description (syntactic and semantic) of a programming language.

This technique involves changing the grammar to generate a new *parser* and especially change the interpreter of the AST whenever the syntax changes. Although there are other techniques proposed in the domain of *Domain-Specific Languages* (DSL) [Mernik *et al.*, 2005] which directly uses syntactic constructs and mechanisms of a host language, and therefore it is more flexible and suits better to constant changes. This technique was used in SCL [Fabresse *et al.*, 2008 ; Fabresse, 2007]. A disadvantage is that it prevents use of special syntax construct like our & operator. Moreover, we consider SMALLTALK syntax to be inappropriate for structural descriptions because it is basically nothing more than a list of message sends.

To eliminate the "*change-generate*" loop necessary for static grammar specification, we decided to use PetitParser framework [Renggli *et al.*, 2010] for building a parser and to develop a custom interpreter. PetitParser combines ideas from scannerless parsing, parser combinators, parsing expression

[1] `www.pharo-project.org`
[2] More exactly in a subset of SMALLTALK called Slang.
[3] like SMACC, `http://www.refactory.com/Software/SmaCC/`

grammars and packrat parsers to model grammars and parsers as objects that can be reconfigured dynamically [Renggli *et al.*, 2010]. This gives us the freedom of syntax evolution without the need to regenerate AST every time the syntax changes.

For the interpreter, we chose to use the very effective AST *visitor pattern* [Gamma *et al.*, 1995a; Buschmann *et al.*, 2008] which also offers great flexibility for the evolution of the syntax.

7.3 Bootstrap Implementation

The bootstrap implementation of COMPO is based on the representation of components and descriptors. There are several requirements the implementation should handle:

Req.1 It should correspond to the basic meta-model (*see* Figure 7.1) of COMPO where descriptor DESCRIPTOR inherits from descriptor COMPONENT and it is an instance of itself (*i.e.* it is both descriptor and meta-descriptor). Descriptor COMPONENT is a root of descriptor-based inheritance and it is an instance of descriptor DESCRIPTOR.

Req.2 It has to respect the parallel hierarchy rule of SMALLTALK which is: meta-classes are implicit and automatically created in SMALLTALK. It is therefore not possible to construct a meta-class which inherits from class

Req.3 It should take advantage of SMALLTALK classes management and make it possible to use standard tools of the SMALLTALK environment to handle descriptor as they were regular classes.

Req.4 It should affect the SMALLTALK's meta-model as less as possible.

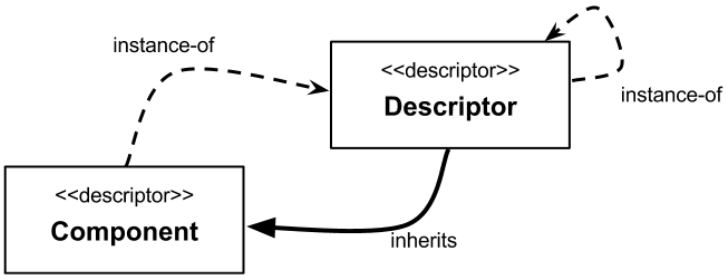


Figure 7.1 : Zoom in to the relation between `Component` and `Descriptor` descriptors

The current implementation of COMPO's core defines class `Descriptor` to represent descriptors and `Component` to represent components. Both are implemented as sub-classes of SMALLTALK-classes: `Object` and `Class`, respectively. Figure 7.2 shows their integration into SMALLTALK's meta-model. This integration makes COMPO components and descriptors manageable inside Pharo SMALLTALK environment. For example, one can use basic inspecting tool, the *Inspector*. `Descriptor` being defined as a sub-class of SMALLTALK-class `Class` enables us to benefit from class management and maintenance capabilities provided by the environment. For example, all descriptors are "browsable" with the standard *SystemBrowser* tool. Such implementation raised few issues we discuss below.

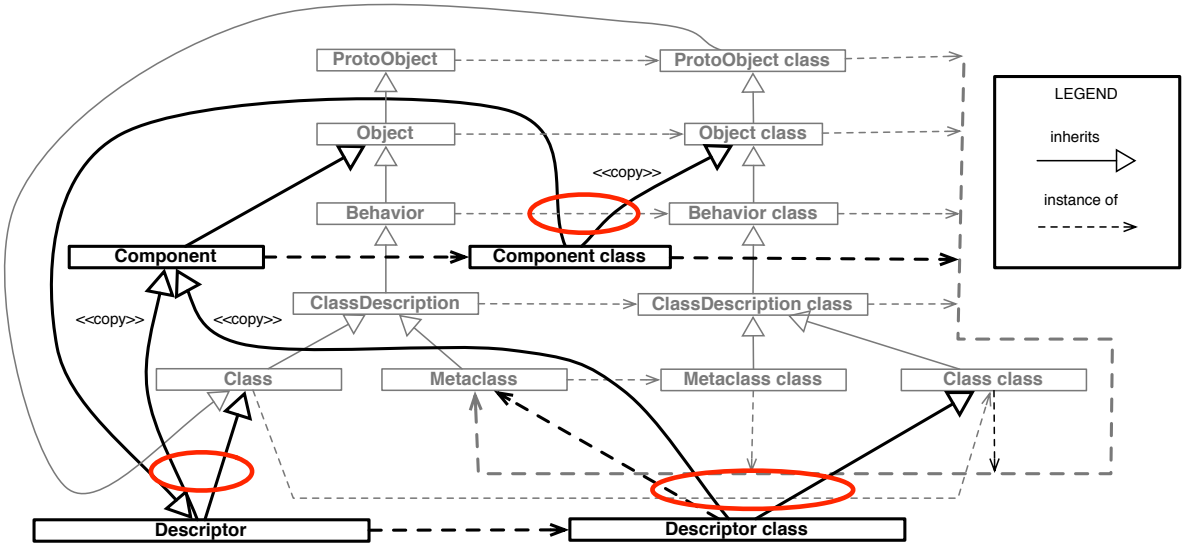


Figure 7.2 : Integrating COMPO's meta-model into SMALLTALK's meta-model.

SMALLTALK uses parallel hierarchy where each class is a unique instance of its anonymous meta-class. Meta-classes are accessible by sending message `class`. The parallel hierarchy rule of SMALLTALK says that when a class inherits from another class (from a super-class), then the meta-class of the class inherits from the meta-class of the super-class. For example, SMALLTALK automatically creates the meta-class `Component class` which, according to parallel hierarchy rules, extends meta-class `Object class`.

One of the problems we challenged during the implementation is the fact that SMALLTALK supports single-inheritance only. The meta-model shown in Figure 7.1 says that `Descriptor` inherits from `Component`, but as it is said above, we implement `Descriptor` as a sub-class of SMALLTALK-classes (`Class`). Consequently `Descriptor` should have two parents and multiple-inheritance[4] is needed. Concretely, there are three critical points, where multiple-inheritance is needed, marked with red ellipses in Figure 7.2:

- `Descriptor` should inherit from SMALLTALK-class `Class` and from class `Component`, to keep all benefits of SMALLTALK's classes management and in the same time to implement the meta-model design of COMPO.

  Solution: we simulate the multiple inheritance by automated copying attributes and methods from `Component` to `Descriptor`;

[4] Although there is a solution based on single-inheritance, the solution introduces an issue when distinguishing components/descriptors from objects/classes in the implementation level and it makes Pharo VM unstable when extending class `Component`.

- The automatically created SMALLTALK-meta-class `Component class` should inherit from SMALLTALK-meta-class `Object class` and from class `Descriptor`, to implement the fact that `Component` is an instance of `Descriptor`.

  Solution: we set the super-class of `Component class` to `Object class` and simulate the multiple inheritance by copying attributes and methods from `Object class` to `Component class` and from `ProtoObject class` to `Component class`.

- Another problem we encountered is the implementation of `Descriptor` as an instance of itself. SMALLTALK-class `Descriptor` is a unique instance of SMALLTALK-meta-class `Descriptor class`, which is automatically created as a sub-class of SMALLTALK-meta-class `Class class` (parallel hierarchy rule of SMALLTALK) and therefore it does not have the same structure as `Descriptor` class.

  Solution: we set the super-class of `Descriptor class` to `Component` and we simulate the multiple inheritance by copying attributes and methods from `Component` to `Descriptor class`.

Indeed, to simulate multiple inheritance by coping attributes and methods requires manual response. When one of the parents (`Component`, `Object class` and `ProtoObject class`) evolves, classes `Descriptor`, `Component class` and `Descriptor class` have to be manually updated. Fortunately, these parent classes are not changed frequently, in fact they remain unchanged for most of the time.

The chosen integration of classes `Descriptor` (to represent descriptors) and `Component` (to represent components) causes that **the following assertions are true**:

- class `Descriptor` inherits class `Component`
- class `Descriptor` is a *kind-of*[5] `Descriptor`, because class `Descriptor` and its meta-class `Descriptor class` have the same structure and the same methods. Because `Descriptor` inherits `Component` and it is a *kind-of* itself, it is also a *kind-of* `Component`.
- class `Component` is a *kind-of* `Descriptor` (indirectly via `Component class`) thus it is also a *kind-of* `Component`.
- every sub-class of `Component` is a *kind-of* `Descriptor` (including classes `Descriptor` and `Descriptor class`) and thus also a *kind-of* `Component`.
- every sub-class of `Descriptor` is a *kind-of* `Descriptor class` and thus it is also a *kind-of* `Descriptor` and a *kind-of* `Component`.

7.4 The implemented model

The implemented model conforms to the meta-model presented in reflection chapter (Chapter 5.5 and Figure 5.3). Figure 7.3 shows a UML class diagram of realizations of all core component concepts

[5] "X is *kind-of* Y" is an expression returning true if X is an instance of Y or of one of its sub-classes.

and their relations. The diagram also shows relations the realizations have with standard SMALLTALK classes.

The two central classes `Descriptor` and `Component` define structure and behavior of descriptors and components respectively. Class `Descriptor` declares three attributes of kind dictionary to reference services, port-descriptions and bind-descriptions.

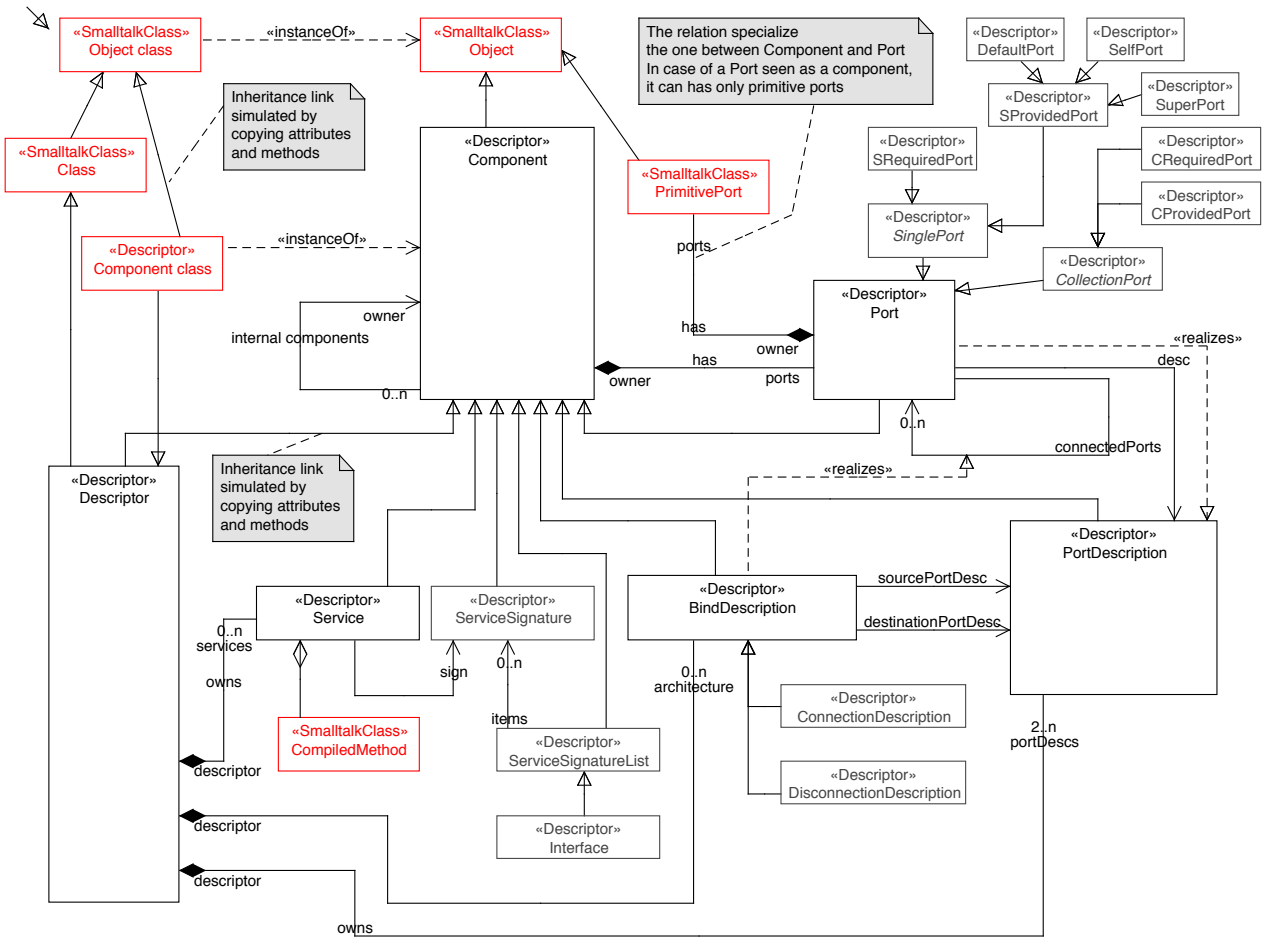


Figure 7.3 : A UML model of COMPO implemented in SMALLTALK

Classes `Port` and `PortDescription` participate in *UML realization relationship* which is a relationship between two model elements, in which one model element (the client) realizes the behavior that the other model element (the supplier) specifies. Ports realize port-descriptions defined by descriptors.

Class `Component` (realizing descriptor COMPONENT, *cf.* 5.1) introduces a dictionary of ports[6].

[6]similarly to SMALLTALK-`Behavior`, which introduces the method dictionary [Black *et al.*, 2009]

A component is constituted of ports (`Port`) and possibly of collection ports (`CollectionPort`). A port is described by a port-description (`PortDescription`) (defined in a descriptor) which defines its name, role, visibility and its interface (`Interface`), *i.e.* a set of service signatures (`ServiceSignatureList`). We distinguish single required ports (`SRequiredPort`) and single provided ports (`SProvidedPort`) and three specific ports that are `self` (`SelfPort`), `super` (`SuperPort`) and `default` (`DefaultPort`). We also distinguish collection required ports (`CRequiredPort`) and collection provided ports (`CProvidedPort`).

Class `BindDescription` describes connection between ports. It defines two attributes to reference instances of `PortDescription`. We distinguish two kinds of bind-description: `ConnectionDescription` and `DisconnectionDescription`. Class `Port` realizes bind-descriptions with attribute `connectedPorts` to store references to other ports and thus making connections.

Services are represented by class `Service`. Each service is associated with a service signature (`ServiceSignature`) which defines its selector and parameters names, for example `sum(a,b)`. For each service there is exactly one SMALLTALK method. The parameters of such method are automatically mapped to the `args` port of the component representing the service. An automatic (and transparent for the user) mapping of a service signature from COMPO to a SMALLTALK's method selector is based on naming convention, for example the following service signature `sum(a,b)` is mapped to a SMALLTALK method with selector `cs__sum__par01:par02:` (the `cs__` prefix stands for Compo-Service). When a compo service invokes the primitive `<primitive_execute>` (see Liisting 5.11 in Section 5.7) it actually executes the associated SMALLTALK method.

7.5 Services invocation implementation

The mechanism of the service invocation was described by the algorithms 2 and 3 (*see* page 97) in Chapter 3.

To illustrate the treatment of a service invocation, let us place ourselves in the context of the example shown in Figure 3.6 where the instance `calculator` of descriptor CALC is connected to an instance of descriptor SOMERANDOMGENERATOR. In this context, consider the following service invocation issued via the port `randGen` component `calculator`.

```
randGen.getRandVal(1);
```

COMPO's interpreter transforms this service invocation to the following SMALLTALK code (note that the `cs__<name>` prefix denotes the SMALLTALK method associated to a COMPO service named `<name>`):

```
port := (self cs__getPortNamed: #randGen).
port cs__invoke__par01: (ServiceInvocation
                          selector: #getRandVal
                          arguments: { 1. }
                          sentThrough: port ).
```

The object (instance of class `Port`) realizing the component representing the port `randGen` implements the method `cs__invoke__par01` associated with the service `invoke` defined by descriptor PORT (*see* Listing 5.5).

Choice 38 says that a port is a component having primitive ports. Therefore an instance of the class `Port` realizing the PORT descriptor own instances of class `PrimitivePort` to represent its own ports. For example the `default` port of an instance of descriptor CALC (*see* Listing 3.4) is realized as an instance of class `Port`. Descriptor PORT defines that ports have, for example, primitive port `connectedPorts`. Thus an instance of class `Port` references an instance of class `PrimitivePort` under the key "`connectedPorts`" of its ports dictionary. Primitive ports implement the behavior of standard ports, as stated by Definition 20.

In the `invoke` service of the descriptor PORT the primitive `<primitive_invoke>` is called. Actually it is a method of the class `Port` representing ports. The method is redefined in sub-classes of `Port` class so that each kind of port specifically addresses the invocations it receives. In case of collection ports, the same processing is preformed for each port in the collection. Listing 7.1 and 7.2 show the method code for provided and required ports:

```
"Code of the primitive_invoke: method defined in class Port"
RequiredPort>>primitive_invoke: aServiceInvocation
   | res receiverObject |
   "Compute the receiver object."
   (self cs__isConnected) ifTrue:[
      "the primitive port connectedPorts will be the receiver"
      receiverObject := self ports at: #connectedPorts.
   ] ifFalse: [
      (self cs__isDelegated) ifTrue: [
         "the primitive port delegatedPorts will be the receiver"
         receiverObject := self ports at: #delegatedPorts.
      ] ifFalse: [
         "stop the execution and throw an error"
         self error: self name, ' : not connected or delegated'
      ].
   ]
   "Transmit the invocation to the receiver object."
   res := receiverObject primitive_invoke: aServiceInvocation.
   "Before returning the result of the invocation, we check the returned value."
   "It has to be of kind Port, otherwise we set the result to reference to self"
   (res isKindOf: Port) ifFalse: [ res := self ].
   "the result is returned"
   ^res
```

LISTING 7.1 : The `<primitive_invoke>` method of the class `RequiredPort`.

```
"Code of the primitive_invoke: method defined in class Port"
ProvidedPort>>primitive_invoke: aServiceInvocation
   | res receiverObject |
   "verify if the owner component implements the requested service"
   (self cs__getOwner respondsTo: (aServiceInvocation smalltalkSel))
      ifTrue: [
         "look up the service"
         |service|
         service := self lookupService: (aServiceInvocation selector)
                                 arity:  (aServiceInvocation args size).
         "set up the arguments"
         self connectArgs: (aServiceInvocation args) forService: service.
         "execute the service"
         res := service cs__execute.
         self disconnectArgs: (aServiceInvocation args) forService: service.
      ] ifFalse: [
         (self cs__isDelegated) ifTrue:[
            "the primitive port delegatedPorts should handle the invocation"
            res := (self ports at: #delegatedPorts) primitive_invoke: aServiceInvocation.
         ] ifFalse: [
            "stop the execution and throw an error"
            self error: self name, ' : not implemented or delegated'.
         ]
      ]
   "Before returning the result of the invocation, we check the returned value."
   "It has to be of kind Port, otherwise we set the result to reference to self"
   (res isKindOf: Port) ifFalse: [ res := self ].
   "the result is returned"
   ^res
```

LISTING 7.2 : The `<primitive_invoke>` method of the class `ProvidedPort`.

7.6 Connection mechanism implementation

As we saw above, the connections are realized by references between ports. Depending on the nature of the port, the service `connectTo` (resp. the associated SMALLTALK method `cs__connecTo__par01:`) of descriptor PORT can establish regular and delegation connections. The **`connect-to`** statement is actually a syntax sugar for `connectTo` service, as we show in Listing 7.3.

In the code of the `connectTo` service the descriptor PORT calls `<primitive_connectTo>` which is a method implemented by class `Port` realizing descriptor PORT. The `<primitive_connectTo>` method calls the `<primitive_connectTo>` of its primitive port `connectTo` (remember ports are components, instance of descriptor PORT, having primitive ports). Primitive ports are instances of class `PrimitivePort` which implements the `<primitive_connectTo>` method as it is shown in Listing 7.4.

Having a solution where components are connected via theirs ports, we consider connections between ports as primitive entities (references), and do not need to reify connections. This entails no limitation regarding the capability to experiment with various kind of connections [Mehta *et al.*, 2000] because our model makes it possible to define new kinds of ports (*see* Section 5.6) and because of the capability it offers to put an adapter component between any components.

```
calculator := Calc.new();
rng := SomeRandomGenerator.new();

/*connecting the ports randGen and default with statement connect-to*/
connect randGen@calculator to default@rng;

/*connecting the ports randGen and default with the connectTo service*/
randGen := calculator.getPortNamed('randGen');
&randGen.connectTo(rng.getPortNamed('default'));
```

LISTING 7.3 : Connecting ports in COMPO

```
"Code of the primitive_invoke: method defined in class Port"
PrimitivePort>>primitive_connectTo: port
   "store the port reference to set of connectedPorts"
   self connectedPorts add: port.
```

LISTING 7.4 : The `primitive_connectTo` method of the class `PrimitivePort`.

7.7 Inheritance implementation

The inheritance mechanism uses SMALLTALK's class inheritance to implement extension and specialization operations for both the structure and behavior of descriptors. A sub-descriptor is implemented as a sub-class of the class representing its super-descriptor.

Extension and specialization of the structure of a component is realized as the modification of the corresponding descriptions (port-descriptions and bind-descriptions) maintained by a descriptor of the component.

For example Listing 7.5 shows an example where descriptor B extends descriptor A with a new required port named `rb`. In the prototype implementation, these descriptors are realized as classes `A` and `B`. Class `A` is a sub-class of class `Component` and class `B` is a sub-class of class `A`.

```
Descriptor A extends Component {}
Descriptor B extends A { requires : { rb : * } }
```

LISTING 7.5 : This COMPO example is implemented with SMALLTALK code in Listing 7.6

Extension and specialization of the behavior leads to addition and specialization of services and their associated SMALLTALK methods. For example, when a sub-descriptor specializes service `foo` of its super descriptor, then a component representing service `foo` is connected to the `services` of the sub-descriptor and the class realizing the sub-descriptor defines method `cs__foo`. The service invocation mechanism of COMPO ensures the correct execution.

```
"Sub-classes of class Component realizes its sub-descriptors"
Component subclass: #A category: 'playground'.
A subclass: #B category: 'playground'.
"Introducing the new required port in B"
B cs__addPortDescription_par01: (PortDescription name: #rb
                                                  role: #required
                                                  visibility: #external
                                                  interface: #* ).
```

LISTING 7.6 : SMALLTALK implementaion of the code from Listing 7.5

7.8 Instantiation mechanism implementation

The instantiation mechanism of COMPO was described in Chapter 3 in Section 3.3.1. In COMPO, descriptors define the structure of components. In the allocation phase of the instantiation mechanism, we analyze descriptor's external and internal contract, *i.e.* the ports it defines, and for each port the mechanism allocates a memory space. The structure and the amount of the memory needed for each port is defined by class `Port` resp. its sub-classes. The initialization phase happens in two steps. During the first step we set the references associating each port with its corresponding port description (*i.e.* references to instances of class `PortDescription`). The second step works with the architecture section of descriptors which describe connections between ports of the created component and ports of internal components. We process each connection description, *i.e.* evaluate both port-address expressions and then we set the binding reference between ports.

The instantiation mechanism is implemented by service `new` of descriptor DESCRIPTOR. In the code of this service, COMPO calls primitive `<newC>` which is a SMALLTALK method implemented by class `Descriptor`. The method creates and initializes new instances (new components) and returns a reference to the default port of a new component. In fact, this technique can be widespread to integrate all SMALLTALK objects, which will then behave as primitive components providing all methods they define (seen as COMPO services) through a unique provided port. Thus SMALLTALK-objects are seen as primitive COMPO-components and they are usable in COMPO, but as components. This will make it possible to reuse SMALLTALK class library.

Integration of SMALLTALK objects The integration of SMALLTALK objects requires modifying the instantiation mechanism of SMALLTALK. When instantiating a class, it must return a reference to the `default` port of the newly created instance and not to the instance directly.

In general, it would have to redefine the `basicNew` in class `Behavior` that creates an instance but it is impossible because it is a primitive of the Pharo's virtual machine. Note that the redefinition of the `new` implemented in the class `Behavior` does not achieve our goal because many classes override this method and use the `basicNew` directly . To implement our instantiation mechanism, we define the following two methods:

- `newC` (shortcut of *newComponent*) in the class `Object class` that allows to instantiate a component from a class and returns a reference to the `default` port of this component,

- `defaultPort` which provides the default port of any SMALLTALK object. The interface port is initialized with all the signatures (a selector contains the arity of the method in SMALLTALK) messages that this object can respond.

In the following example, we show the use of these methods to integrate basic SMALLTALK objects in the form of COMPO components:

```
/* Instantiation of the class OrderedCollection */
col := OrderedCollection.new();

/* the temporal required port col is connected to
   a provided port named 'default' */
col.size();                          /* invocation of service size /*

/* literals are automatically treated as components */
aPrimitiveComponent := 1.

/* the temporal required port aPrimitiveComponent is connected to
   a provided port named 'default' */"
aPrimitiveComponent.odd();         /* invocation of service odd */
```

In the case of literals such as integers or strings, the COMPO programmer should store the reference to their default port using the `defaultPort`. In fact, the literal does not benefit directly from the `newC` defined in class `Object class` because they are never really instantiated but specifically addressed by the Pharo virtual machine. To standardize the vision offered to the COMPO programmer, we also defined the `newC` in class `Object` as follows:

```
Object>>newC
   ^self defaultPort
```

This feature enables COMPO's interpreter to create primitive components and standard components in the same manner by invoking the **`newC`** primitive.

7.9 Toward a graphical development environment

Figure 7.4 shows a screenshot of a visual development tool for browsing and writing new COMPO descriptors. This prototype tool currently allows an architect to browse descriptors in a library. Once a descriptor is selected, the users may edit its code. All the changes are immediately propagated to a graphical visualization of an instance of the descriptor. Then, it is possible to enter the COMPO code in the box at the bottom left to invoke the services of the instance. The results can be displayed in the window at the bottom right which is the standard output. This browser/editor environment is a step towards a graphical development environment as it is a simple useful tool to understand and put into practice for developing new components prototypes.

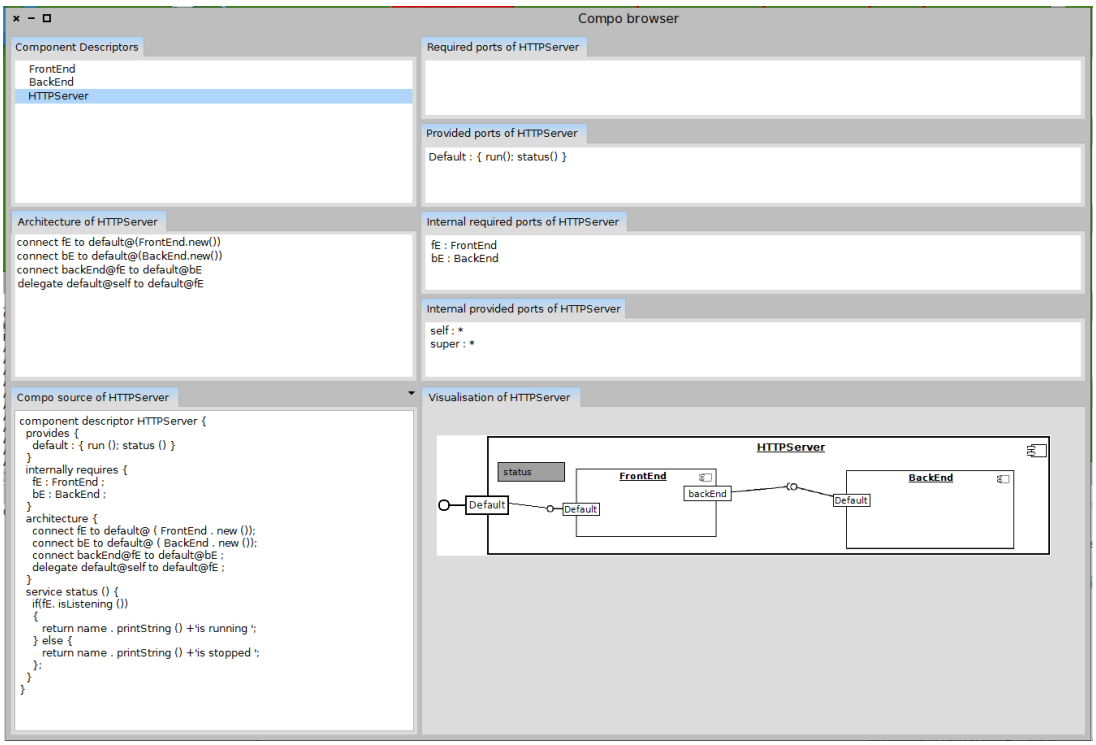


Figure 7.4 : Screenshot of the Compo browser, a step towards visual development

7.10 Summary

In this chapter, we presented a prototype of COMPO written in Pharo SMALLTALK. We also presented the arguments that led us to choose the SMALLTALK language and our implementation choices. When specifying the prototype, we have:

- addressed the problem of bootstrap implementation which requires that the class representing descriptor DESCRIPTOR inherits from SMALLTALK-class `Class`
- described the implemented UML model
- integrated SMALLTALK objects and COMPO components by providing the primitive **`newC`**.
- explained the implementation of the service invocation
- illustrated assembly of components using connections
- shown, finally, a visual tool (COMPO browse) for writing new descriptors in COMPO

The point of improvement of the prototype is certainly its effectiveness. Indeed, we clearly preferred evolution over efficiency. It is difficult to quantify the effectiveness of the current prototype but the implementation we have chosen for the service invocation (explicit delegation between objects representing the ports) suggests that the performance of the implementation could be improved.

CHAPTER 8

Conclusion

THIS thesis contributes to the domain of component-based software engineering (CBSE) by proposing a reflective component-oriented language named COMPO which enables users to describe and continuously implement component-based architectures. On behalf of the work, we have studied many component-based approaches ranging from architecture description languages over component frameworks to component-oriented programming languages. The study convinced us that the potential problems, like the non-conformance between an architecture design and its implementation, raising from the fact that most of component-based approaches separate design and implementation stage could be overcame when a component-oriented programming language is used. Embedding architecture into an implementation language lets architects specify the architecture of a system in much more detail, and its presence in the source code provides developers with a constant awareness of architectural issues. However, while the conformance between design and implementation is well addressed by the existing component-oriented languages, a support for software evolution and for model driven development has not yet been well addressed by these approaches.

In our work, we have partially answered to that issue by designing a reflective component-oriented language with an inheritance mechanism for structural and behavioral reuse. We believe that reflection and inheritance are the key factors directly supporting evolution and maintenance of software developed in COMPO. Reflection simply opens the possibility that architectures, implementations and transformations can all be written at the component level and using a unique language.

The study made in Chapter 2 helped us to identify the core concepts and mechanisms of CBSE and to built COMPO language on top of them. The core mechanisms: instantiation, composition, service invocation and substitution together with the identified concepts: components and their descriptors, ports, connections and services, provide architecture description constructs, so that developers can specify an architecture during design and then fill in the architecture with COMPO implementation.

Having the architecture part well described, it is possible to write the implementation of services in various (future) COLs or even in an OOP language. Thus, we also support the idea that it is interesting to implement models in various contexts.

Communication protocol as presented in COMPO is based on the idea that the only way two components can communicate is by sending a service invocation through a connection between their ports. Existing component-oriented languages support hierarchical design, *i.e.* to describe architecture in terms of components (*composites*) which are composed of other components (*internal component*) which are composed of other components, etc. However, the communication between different levels of the hierarchical design has not been addressed and thus there was no protocol saying how these internal components communicate with their owning composite and vice verse. A one of contributions of this work is that thanks to the concept of *internal required ports* we were able to preserve this statement even for communication between a composite and its internal components. Such a communication protocol does not introduce an additional mechanism or concept to the language and it enforces communication integrity because all the communication is well described by connections.

The inheritance mechanism proposes an innovative reuse scheme in the context of CBSE by bringing an objects like inheritance capability to components descriptions. Inheritance in COMPO promotes modeling power with covariant specializations via the *extends* statement. A new descriptor (a *sub-descriptor*) can be defined on the base of an existing descriptor by extending or specializing its definition. Indeed, covariant specialization has advantages but also drawbacks, for example it is hard to ensure substitutability between instances of sub- and super-descriptors. We choose a coherent policy comparing its advantages and drawbacks. We believe that developers are much more interested in specializing and extending at the same time provisions and requirements of a component, and less on substitutability, which they can manage manually (by satisfying additional requirements, if needed). Because we were unable to ensure type-safe substitution, we have proposed a substitution mechanism based on run-time checks which if used properly, preserves the safety of substitutions. Thus, sub-descriptors may: (1) introduce new ports or extend interfaces of inherited ports, (2) introduce new services and override inherited services and (3) extend and specialize the inherited architecture description. The ability to inherit existing architectures make it possible to capitalize on good designs where well-established architecture styles or patterns are applied.

As far as we know COMPO is the first fully reflective component-oriented language with core component concepts reified in terms of components. In Chapter 5 , we have proposed the component-based model compliant to its meta-model and the component-based meta-model compliant to itself. There is only one kind of entity, component: a descriptor is a component and a meta-descriptor is a descriptor whose instances are descriptors. This allows a simplification and economy of concepts, which are thus more powerful and general. Reflection makes the language uniformly accessible by users who can introspect the underlying structure and behavior of the platform and also adaptation them if needed. Reflection also allows to experience the impact of adding new mechanisms at both the architectural and implementation levels. For example: to define different control facilities for components such as non-functional aspects or to define trade-offs such as degree of configurability vs performance and space consumption.

Future work COMPO in its today's state is a research laboratory and does not yet embed all capabilities offered by existing ADLs or COLs, but we believe that there is no conceptual lock to the integration of new concepts. For example, dynamic software architectures represent one encouraging approach to mitigate an important class of safety- and mission-critical software systems, such as: telephone dynamic update in high availability public information systems. Dynamic software architectures [Baresi *et al.*, 2004 ; Barais *et al.*, 2008] are software architecture descriptions that include not only the description of fixed (*i.e.* static) parts, but also the description of changing (*i.e.* dynamic) parts. ArchJava or Darwin have shown that the dynamic aspect of architectures can be captured at the language level enlarging the spectrum of problems such languages can be used for. Therefore, in future, we would like to extend COMPO with the ability to describe dynamic architectures.

In Section 5.6 we tried to show that it is possible to define new communication protocols or lookup policies by creating new kinds of ports. This makes it possible to achieve scenarios similar to the ones in the object-oriented context, where first-class references are introduced [Arnaud *et al.*, 2010] or the ones where custom lookup objects are needed [Vraný *et al.*, 2012]. Thus, it would be nice to study all the possibilities which come with first-class ports. Moreover the meta-model architecture of COMPO makes it possible to define new kinds of descriptors. It should be possible to define a new kind of "deployment-location-aware" descriptors whose instances will be descriptors aware of the location where they can be instantiated. This in turn would make it possible to explicitly describe an architecture which is distributed over multiple execution nodes. In addition, to deploy components a packaging tool, similar to OSGi bundles or similar things, would be needed.

Another prospective work is to design a (visual) graphical development environment. In such development environment it should be possible to define new descriptors graphically in way that is similar to the one of the applications for designing UML diagrams. This would also require to integrate the notion of "properties", so another components could listen for "value change" events of the properties. In fact, properties for components have been integrated in SCL, thus we just have to adapt the SCL's solution for COMPO.

To optimize programs efficiency is another remaining tasks. Many solutions do exist [Chiba, 1997]. For COMPO, we already do have an initial idea for a pure virtual-machine where only entities managed by the machine will be components. This should enhance the language performance in comparison to the current implementation hosted within a third party virtual machine.

APPENDIX A

Grammar

EBNF form, quick help:

```
( start-group-symbol end-group-symbol )
[ start-option-symbol end-option-symbol ]
{ start-repeat-symbol end-repeat-symbol }
| definition-separator-symbol
* repetition-symbol
- except-symbol
, concatenate-symbol
= defining-symbol
; terminator-symbol
```

A.1 Lexan rules

```
Character = ? Any Unicode character ?;
WhitespaceCharacter = ? Any space, newline or horizontal tab character ?;
DecimalDigit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";
Letter = "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" | "M"
       | "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z"
       | "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m"
       | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z";
CommentCharacter = Character - '"';  (* Any character other than a double quote *)
Comment = '"', {CommentCharacter}, '"';
OptionalWhitespace = {WhitespaceCharacter | Comment};
Whitespace = (WhitespaceCharacter | Comment), OptionalWhitespace;
LetterOrDigit = DecimalDigit
              | Letter;
```

```
Identifier = (Letter | "_"), {(LetterOrDigit | "_")};
Reference = Identifier;
ConstantReference = "nil"
                  | "false"
                  | "true";
PseudoVariableReference = "self" | "super" | "myPorts" | "default" | "myConnections";
ReservedIdentifier = PseudoVariableReference
                   | ConstantReference;
BindableIdentifier = Identifier - ReservedIdentifier;
StdMessageSelector = Identifier;
BinarySelectorChar = "~" | "!" | "@" | "[PERCENT]" | "&" | "*" | "-" | "+" | "=" | "|" | "<" | ">" | "
BinaryMessageSelector = BinarySelectorChar, [BinarySelectorChar];
IntegerLiteral = ["-"], UnsignedIntegerLiteral;
UnsignedIntegerLiteral = DecimalIntegerLiteral
                       | Radix, "r", BaseNIntegerLiteral;
DecimalIntegerLiteral = DecimalDigit, {DecimalDigit};
Radix = DecimalIntegerLiteral;
BaseNIntegerLiteral = LetterOrDigit, {LetterOrDigit};
ScaledDecimalLiteral = ["-"], DecimalIntegerLiteral, [".", DecimalIntegerLiteral], "s", [DecimalIntege

FloatingPointLiteral = ["-"], DecimalIntegerLiteral, (".", DecimalIntegerLiteral, [Exponent] | Exponer
Exponent = ("e" | "d" | "q"), [["-"], DecimalIntegerLiteral];
CharacterLiteral = "[DOLAR]", Character;
StringLiteral = "'", {StringLiteralCharacter | "''"}, "'"; (* To embed a "'" character in a String lit
StringLiteralCharacter = Character - "'";  (* Any character other than a single quote *)
SymbolInArrayLiteral = StdMessageSelector - ConstantReference
                     | BinaryMessageSelector;
SymbolLiteral = "#", (SymbolInArrayLiteral | ConstantReference | StringLiteral);
ArrayLiteral = ObjectArrayLiteral
             | ByteArrayLiteral;
ObjectArrayLiteral = "#", NestedObjectArrayLiteral;
NestedObjectArrayLiteral = "(", OptionalWhitespace, [LiteralArrayElement, {Whitespace, LiteralArrayEle
LiteralArrayElement = Literal - BlockLiteral
                    | NestedObjectArrayLiteral
                    | SymbolInArrayLiteral
                    | ConstantReference;
ByteArrayLiteral = "#[", OptionalWhitespace, [UnsignedIntegerLiteral, {Whitespace, UnsignedIntegerLite

DereferenceLiteral = "&" , Reference ;
(* Operator "&" enables to see ports as components, semantics is: (&aPort).isConnected() == myPorts[my
CollectionPortLiteral = Reference , "[" , Expression , "]" ;
PortAddressLiteral = Reference , "@" , Reference ;
```

A.2 Parser rules

```
FormalBlockArgumentDeclaration = ":", BindableIdentifier;
FormalBlockArgumentDeclarationList = FormalBlockArgumentDeclaration, {Whitespace, FormalBlockArgumentD
BlockLiteral = "[", [OptionalWhitespace, FormalBlockArgumentDeclarationList, OptionalWhitespace, "|"],
Literal = ConstantReference
```

```
        | IntegerLiteral
        | ScaledDecimalLiteral
        | FloatingPointLiteral
        | CharacterLiteral
        | StringLiteral
        | SymbolLiteral
        | DerefenceLiteral
        | CollectionPortLiteral
        | ArrayLiteral
        | BlockLiteral;
NestedExpression = "(", Statement, OptionalWhitespace, ")";
Operand = Literal
        | Reference
        | NestedExpression;
FormalStdMsgArgumentDeclaration = [BindableIdentifier , ":"] , StdMessageArgument;
FormalStdMsgArgumentDeclarationList = FormalStdMsgArgumentDeclaration , { OptionalWhitespace, ",", Optional
StdMessageArgument = BinaryMessageOperand, BinaryMessageChain;
StdMessage = StdMessageSelector, "(" , OptionalWhitespace, [ FormalStdMsgArgumentDeclarationList, OptionalW
StdMessageChain = {OptionalWhitespace, UnaryMessage};
BinaryMessageOperand = Operand, UnaryMessageChain;
BinaryMessage = BinaryMessageSelector, OptionalWhitespace, BinaryMessageOperand;
BinaryMessageChain = {OptionalWhitespace, BinaryMessage};
MessageChain = ".", StdMessage, StdMessageChain, BinaryMessageChain
             | OptionalWhitespace, BinaryMessage, BinaryMessageChain;
CascadedMessage = ",", OptionalWhitespace, MessageChain;
Expression = Operand, [MessageChain, {OptionalWhitespace, CascadedMessage}];
AssignmentOperation = OptionalWhitespace, BindableIdentifier, OptionalWhitespace, ":=";
Statement = {AssignmentOperation}, OptionalWhitespace, Expression;
MethodReturnOperator = OptionalWhitespace, "return";
FinalStatement = [MethodReturnOperator], Statement;
LocalVariableDeclarationList = OptionalWhitespace, "|", OptionalWhitespace, [BindableIdentifier, {Whitespace
ExecutableCode = [LocalVariableDeclarationList], [{Statement, OptionalWhitespace, ";"}, FinalStatement, [";

CompoIdent = Identifier;
CompoServiceSign = CompoIdent , "(" , [{ CompoIdent , "," }] , [CompoIdent] , ")" ;
ServiceSignsList = "{", [{CompoServiceSign , ";"}], [CompoServiceSign] , "}"
Connection = "connect", PortAddressLiteral, "to" , PortAddressLiteral;
Disconnection = "disconnet", PortAddressLiteral, "from" , PortAddressLiteral;
PortDecl = ["atomic" ], CompoIdent , ":" , (CompoIdent | ServiceSignsList);
ExProvisions = ["externally"] , "provides", CompoIdent , "{", [{PortDecl, ";"}], [PortDecl] , "}";
ExRequirements = ["externally"] , "requires", CompoIdent , "{", [{PortDecl, ";"}], [PortDecl] , "}";
InProvisions = "internally" , "provides", CompoIdent , "{", [{PortDecl, ";"}], [PortDecl] , "}";
InRequirements = "internally" , "requires", CompoIdent , "{", [{PortDecl, ["inject-with" , CompoIdent], ";"
Services = "service", CompoServiceSign, "{" , ExecutebaleCode , "}";
Contraints = "constraint", CompoServiceSign, "{" , ExecutebaleCode , "}";
Architecture = "architecture", "{", [{(Connection | Disconnection) , ";"}], [(Connection | Disconnection)]
CompoExpr = ExProvisions
          | ExRequirements
          | Services
          | Constraints
```

```
          | InProvisions
          | InRequirements
          | Architecture;
ComponentDecl = "component descriptor", CompoIdent , ["extends" , CompoIdent] , "{", {CompoExpr} , "}"
Interface = "interface", CompoIdent , ["extends" , CompoIdent] , ServiceSignsList;
CompoStart = [{ComponentDecl | Interface}];
```

APPENDIX B

Usage sources

B.1 Collection hierarchy sources

```
Descriptor AbstractSet extends Collection {
   /* new abstract not public service*/
   service getItemAt(index);
   /* new abstract not public service*/
   service areEqual(itemA, itemB);

   service foreachDo(closure) {
      |i|
      for(i:=0;i<self.size();i:=i+1) {
         connect args@closure to default@(self.getItemAt(i));
         closure.execute();
         disconnect args@closure from default@(self.getItemAt(i));
      }
   }

   /* not public */
   service indexOf(item) {
      |i|
      for(i:=0;i<self.size();i:=i+1) {
         if(self.areEqual(self.getItemAt(i), item))
         { return i; }
      }
      return -1;
   }
}
```

LISTING B.1 : The SET descriptor

```
Descriptor Bag extends Collection {
   internally requires {
      items[] : * ;
      tally[] : *
   }
   service size() { return sizeof(items); }
   service add(item) {
      |itemClone i|
      i := self.indexOf(item);
      if(i<0) {
         itemClone := item;
         &items.connectTo(default@itemClone);
         self.zeroTally(i);
         self.addToTally(i);
      }else{
         self.addToTally(i);
      }
   }
   service remove(item) {
      |i|
      i := self.indexOf(item);
      if(i>=0) {
         if(self.getTally() > 1)
         { self.subFromTally(i); }
         else
         {
            &items.disconnect(i);
            self.zeroTally();
         }
      }
   }
   service foreachDo(closure) {
      |i|
      for(i:=0;i<self.size();i:=i+1) {
         connect args@closure to default@(items[i]);
         closure.execute();
         disconnect args@closure from default@(items[i]);
      }
   }
   /* not public */
   service indexOf(item) {
      |i|
      for(i:=0;i<self.size();i:=i+1) {
         if(self.getIdentityHash() == item.getIdentityHash())
         { return i; }
      }
      return -1;
   }
   service getTally(index) { return tally[index]; }
   service zeroTally(index) { tally[index] := 0; }
   service addToTally(index) { tally[index] := tally[index] + 1 }
   service subFromTally(index) { tally[index] := tally[index] - 1 }
}
```

LISTING B.2 : The BAG descriptor

```
Descriptor SequenceableCollection extends Collection
{
   provides {
      accessing : { getIndex(index); setIndex(index,item); };
      removing : { removeIndex(index); };
      testing : { indexOf(item); };
   }
   internally requires
   {
      items[] : *;
   }
   service size() { return sizeof(items); }
   service add(item) {  &items.connectTo(default@item); }
   service remove(item) {
      |i|
      i := self.indexOf(item);
      if(i>=0) { self.removeIndex(i); }
   }
   service removeIndex(index) { &items.disconnect(i); }
   service getIndex(index) { if(self.indexOK(index) { return items[index]; }  }
   service setIndex(index,item) {
      if(self.indexOK(index))
      {
         items.disconnect(index);
         connect items[index] to default@item;
      }
   }
   service foreachDo(closure) {
      |i|
      for(i:=0;i<self.size();i:=i+1) {
         connect args@closure to default@(items[i]);
         closure.execute();
         disconnect args@closure from default@(items[i]);
      }
   }
   service indexOf(item) {
      |i|
      for(i:=0;i<self.size();i:=i+1) {
         if(self.getIdentityHash() == item.getIdentityHash())
         { return i; }
      }
      return -1;
   }
   /* not public */
   service indexOK(index) { return ((index >=0)&(index<self.size())); }
}
```

LISTING B.3 : The SEQUENCEABLECOLLECTION descriptor

```
Descriptor Stack extends SequenceableCollection
{
   provides {
      stackable : {  push(item); pop(); }
   }
   service remove(item) { error('this is a stack, use push-pop'); }
   service indexOf(item) { error('this is a stack, use push-pop'); }
   service removeIndex(index) { error('this is a stack, use push-pop'); }
   service getIndex(index) { error('this is a stack, use push-pop'); }
   service setIndex(index) { error('this is a stack, use push-pop'); }

   service push(item){  self.add(item); }
   service pop(){
      |item top|
      top := self.size()-1;
      connect item to defautl@(items[top]);
      &items.disconnect(top);
      return item;
   }
}
```

LISTING B.4 : The STACK descriptor

```
Descriptor Set extends AbstractSet {
   internally requires {
      items[] : * ;
   }
   service size() { return sizeof(items); }
   service add(item) {
      |itemClone i|
      i := self.indexOf(item);
      if(i<0) {
         itemClone := item;
         &items.connectTo(default@itemClone);
      }
   }
   service remove(item) {
      |i|
      i := self.indexOf(item);
      if(i>=0) { &items.disconnect(i); }
   }
   /* not public */
   service getItemAt(index) {
      |i|
      if(i>=0 & i<self.size()) {
         return items[i];
      }
      else { error('index out of bounds') }
   }
   /* not public */
   service areEqual(itemA, itemB) {
      return itemA == itemB;
   }
}
```

LISTING B.5 : The SET descriptor

```
Descriptor IdentitySet extends AbstractSet {
   requires {
      items[] : * ofKind SetCollPort;
   }
   service size() { return sizeof(items); }
   service add(item) { &items.connectTo(default@item);  }
   service remove(item) {
      |i|
      i := self.indexOf(item);
      if(i>=0) { &items.privatedisconnect(i); }
   }
   /* not public */
   service getItemAt(index) {
      |i|
      if(i>=0 & i<self.size()) {
         return items[i];
      }
      else { error('index out of bounds') }
   }
   /* not public */
   service areEqual(itemA, itemB) {
      return itemA.getIdentityHash() == itemB.getIdentityHash();
   }
}
```

LISTING B.6 : The SET descriptor

```
Descriptor SetCollPort extends CollectionPort {
   service connectTo(port){
      |i|
      i := owner.indexOf(item);
      if(i<0) { super.connectTo(port) }
   }
   service disconnect(index) {}
   service privatedisconnect(index) {super.disconnect(index);}
}
```

LISTING B.7 : The SETCOLLPORT descriptor

B.2 Constraints sources

B.2.1 Pipes&Filters

```
Descriptor SubConstraintTwo extends Constraint
{
    service verify() {
        |retval|
        retval := true;
        intComps := context.getPorts().select([:p |
            &p.isRequired().and([&p.isInternal()]);
        ]);
        intComps.each([:ic |
            ic.getPorts().each([:x |
                if(&x.isRequired().and([&p.isExternal()])) {
                    | count |
                    &x.getConnectedPorts().each([:cp |
                        if(&cp.isRequired().and([&p.isExternal()])) {
                            if(&cp.getOwner() == context.yourself())
                            { retVal := retVal.and([true]); }
                            else
                            { retVal := retVal.and([false]); }
                        }
                    ]);

                }
                if(&x.isProvided().and([&p.isExternal()])) {
                    &x.getConnectedPorts().each([:cp |
                        if(&cp.isRequired().and([&p.isExternal()])) {
                            if(&cp.getOwner().getOwner() == context.yourself())
                            { retVal := retVal.and([true]); }
                            else
                            { retVal := retVal.and([false]); }
                        }
                    ]);
                }
            ]);
         ]);
        return retVal;
    }
}
```

LISTING B.8 : PipeAndFilter, the sub constraint two in COMPO. There is only one internal component having one or more external required ports connected uniquely to the owning composite only (delegated ports). The component have to have one or more external provided ports, each being connected to an internal component of the same hierarchical level, or being not connected at all.

```
Descriptor SubConstraintThree extends Constraint
{
    service verify() {
        |count|
        count := 0;
        intComps := context.getPorts().select([:p |
            &p.isRequired().and([p.isInternal()]);
        ]);
        intComps.each([:ic  |
            ic.getPorts().select([:p | &p.isExternal()]).each([:ep |
                |trueForAll|
                trueForAll := true;
                &ep.getConnectedPorts().each([:cp |
                    if(&cp.getOwner().getOwner() == context.yourself())
                    { trueForAll := trueForAll.and([true]); }
                    else
                    { trueForAll := trueForAll.and([false]); }
                ]);
                if(trueForAll) { count := count + 1 }
            ]);
        ]);
        if(count == (intComps.size() - 2))
        { return true; }
        else
        { return false; }
    }
}
```

LISTING B.9 : PipeAndFilter, the sub constraint three in COMPO. Other internal components (n-2) have external provided and required ports connected to other internal components of the same hierarchical level.

```
Descriptor SubConstraintFour extends Constraint
{
    service verify() {
        conns := context.getDescriptor().getDescribedConnections();
        conns.each([:conn |
            |dest source |
            source := conn.getSourcePortComponent();
            dest := conn.getDestinationPortComponent();
            conns.each([:conn2 |
                if((conn2.getSourcePortComponent() == dest).and([
                    conn2.getDestinationPortComponent() == source]))
                { return false; }
            ]);
        ]);
        return true;
    }
}
```

LISTING B.10 : PipeAndFilter, the sub constraint four in COMPO. Connection between each pair of internal components should go in the same direction, i.e. there are not two connections of opposite direction between each pair.

```
Descriptor SubConstraintFive extends Constraint
{
    service verify() {
        conns := context.getDescriptor().getDescribedConnections();
        conns.each([:conn |
            |destPD sourcePD connsA connsB|
            sourcePD := conn.getSourcePortDescription();
            destPD := conn.getDestinationPortDescription();
            /* for each connection, there are no other two where the former has
             the same source, the later the same destination and the two have
             a common port end. */
            connsA  := conns.select([:conn2 |
                | sourcePD2|
                if((conn == conn2).not())
                {
                    sourcePD2 := conn2.getSourcePortDescription();
                    (sourcePD == sourcePD2);
                }else{ false }
            ]);
            connsB  := conns.select([:conn2 |
                | sourcePD2|
                if((conn == conn2).not())
                {
                    destPD2 := conn2.getDestinationPortDescription();
                    (destPD == destPD2);
                }else{ false }
            ]);
            connsA.each([:cA |
                |destA |
                destA := cA.getDestinationPortDescription();
                connsB.each[:cB |
                    |srcB|
                    srcB := cB.getSourcePortDescription();
                    if(destA.getComponent() == srcB.getComponent())
                    { return false }
                ]);
            ]);
        ]);
        return true;
    }
}
```

LISTING B.11 : PipeAndFilter, the sub constraint five in COMPO. For each pair (A, B) of directly connected internal components, there is not a third component, which is connected to the required ports of A and in the same time to provided ports of B.

List of Figures

List of Tables

List of Listings

Bibliography

[Abadi et Cardelli, 1996] Martin Abadi et Luca Cardelli. *A Theory of Objects*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.

[Aldrich *et al.*, 2002] Jonathan Aldrich, Craig Chambers, et David Notkin. ArchJava: Connecting Software Architecture to Implementation. In *ICSE*, pages 187–197. ACM, 2002.

[Aldrich, 2003] Jonathan Aldrich. *Using Types to Enforce Architectural Structure*. PhD thesis, University of Washington, August 2003.

[Allen et Garlan, 1994] Robert Allen et David Garlan. Formalizing architectural connection. In *Proceedings of the 16th international conference on Software engineering*, ICSE '94, pages 71–80, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[Allen, 1997] Robert John Allen. *A formal approach to software architecture*. PhD thesis, Pittsburgh, PA, USA, 1997. AAI9813815.

[Anantharam, 2001] Parasuram Anantharam. Programming ruby. *SIGSOFT Software Engineering Notes*, 26(4):89–89, 2001.

[Arnaud *et al.*, 2010] Jean-Baptiste Arnaud, Marcus Denker, Stéphane Ducasse, Damien Pollet, Alexandre Bergel, et Mathieu Suen. Read-only execution for dynamic languages. In *Proceedings of the 48th international conference on Objects, models, components, patterns*, TOOLS'10, pages 117–136, Berlin, Heidelberg, 2010. Springer-Verlag.

[Arnout, 2004] Karine Arnout. From patterns to components. ETH Zürich, 2004.

[Barais *et al.*, 2008] Olivier Barais, Anne Françoise Meur, Laurence Duchien, et Julia Lawall. Software architecture evolution. In *Software Evolution*, pages 233–262. Springer Berlin Heidelberg, 2008.

[Baresi *et al.*, 2004] L. Baresi, R. Heckel, S. Thone, et D. Varro. Style-based refinement of dynamic software architectures. In *Software Architecture, 2004. WICSA 2004. Proceedings. Fourth Working IEEE/IFIP Conference on*, pages 155–164, 2004.

[Beugnard *et al.*, 1999] Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau, et Damien Watkins. Making Components Contract Aware. *Computer*, 32(7):38–45, 1999.

[Beugnard et Sadou, 2007] Antoine Beugnard et Salah Sadou. Method overloading and overriding cause distribution transparency and encapsulation flaws. *Journal of Object Technology*, 6(2):31–45, 2007.

[Black *et al.*, 2009] Andrew Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, et Marcus Denker. *Pharo by Example*. Square Bracket Associates, 2009.

[Blair *et al.*, 1998] G. S. Blair, G. Coulson, P. Robin, et M. Papathomas. An architecture for next generation middleware. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, Middleware '98, pages 191–206, London, UK, UK, 1998. Springer-Verlag.

[Blair *et al.*, 2009] G. Blair, N. Bencomo, et R.B. France. Models@ run.time. *Computer*, 42(10):22–27, 2009.

[Blanc *et al.*, 2007] Xavier Blanc, Jérôme Delatour, et Tewfik Ziadi. Benefits of the mde approach for the development of embedded and robotic systems. In *Proceedings of the 2nd National Workshop on Control Architectures of Robots*, CAR'07, pages 124–134, Mai 2007.

[Bobrow *et al.*, 1986] Daniel G. Bobrow, Kenneth Kahn, Gregor Kiczales, Larry Masinter, Mark Stefik, et Frank Zdybel. Commonloops: merging Lisp and object-oriented programming. In *OOPLSA '86: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 17–29, New York, NY, USA, 1986. ACM Press.

[Bouraqadi et Fabresse, 2009] Noury Bouraqadi et Luc Fabresse. Clic: a component model symbiotic with smalltalk. In *procs. of IWST*, New York, NY, USA, 2009. ACM.

[Boyland et Castagna, 1996] John Boyland et Giuseppe Castagna. Type-safe compilation of covariant specialization: A practical case. In *ECOOP '96 — Object-Oriented Programming*, éditeur Pierre Cointe, volume 1098 de *Lecture Notes in Computer Science*, pages 3–25. Springer Berlin Heidelberg, 1996.

[Bracha et Cook, 1990] Gilad Bracha et William Cook. Mixin-Based Inheritance. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications / Proceedings of the European Conference on Object-Oriented Programming*, éditeur Norman Meyrowitz, pages 303–311, Ottawa, Canada, 1990. ACM Press.

[Briand *et al.*, 1999] Lionel C. Briand, John W. Daly, et Jürgen K. Wüst. A Unified Framework for Coupling Measurement in Object-Oriented Systems. *IEEE Trans. Software Eng.*, 25(1):91–121, 1999.

[Briot et Cointe, 1989] J.-P. Briot et P. Cointe. Programming with explicit metaclasses in smalltalk-80. *SIGPLAN Not.*, 24(10):419–431, Septembre 1989.

[Bruneton *et al.*, 2004] E Bruneton, T Coupaye, et J.B. Stefani. The fractal component model. Rapport technique, OW2 Consortium, February 2004.

[Bruneton *et al.*, 2006] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, et Jean-Bernard Stefani. The fractal component model and its support in java: Experiences with auto-adaptive and reconfigurable systems. *Softw. Pract. Exper.*, 36(11-12):1257–1284, Septembre 2006.

[Büchi et Weck, 1998] Martin Büchi et Wolfgang Weck. Compound types for Java. In *OOPSLA'98: Proceedings of the 13th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 362–373, New York, NY, USA, 1998. ACM Press.

[Bures *et al.*, 2006] Tomas Bures, Petr Hnetynka, et Frantisek Plasil. Sofa 2.0: Balancing advanced features in a hierarchical component model. In *Proceedings of the Fourth International Conference on Software Engineering Research, Management and Applications*, pages 40–48, Washington, DC, USA, 2006. IEEE Computer Society.

[Buschmann *et al.*, 2008] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, et M. Stal. *PATTERN-ORIENTED SOFTWARE ARCHITECTURE: A SYSTEM OF PATTERNS*. Numéro sv. 1. Wiley India Pvt. Limited, 2008.

[Capra et Cazzola, 2009] L. Capra et W. Cazzola. *An Introduction to Reflective Petri Nets*, pages 191 – 217. IGI Global, 2009.

[Cardelli, 1997] Luca Cardelli. *The Handbook of Computer Science and Engineering*, chapitre 103, Type Systems, pages 2208–2236. CRC Press, Boca Raton, FL, 1997.

[Carrière *et al.*, 1999] S. Jeromy Carrière, Steven G. Woods, et Rick Kazman. Software architectural transformation. In *Proc. Working Conf. Reverse Engineering (WCRE)*, pages 13–23. IEEE Computer Society Press, 1999.

[Cazzola, 1998] Walter Cazzola. Evaluation of object-oriented reflective models. In *Object-Oriented Technology: ECOOP'98 Workshop Reader*, éditeurs Serge Demeyer et Jan Bosch, volume 1543 de *Lecture Notes in Computer Science*, pages 386–387. Springer Berlin Heidelberg, 1998.

[Cheesman et Daniels, 2000] John Cheesman et John Daniels. *UML components: a simple process for specifying component-based software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[Chiba, 1997] Shigeru Chiba. Implementation techniques for efficient reflective languages. Rapport technique, Departement of Information Science, The University of Tokyo, 1997.

[Cioch *et al.*, 2000] Frank A. Cioch, John M. Brabbs, et Larry Sieh. The impact of software architecture reuse on development processes and standards. *Journal of Systems and Software*, 50(3):221 – 236, 2000.

[Clarke *et al.*, 2001] Michael Clarke, Gordon S. Blair, Geoff Coulson, et Nikos Parlavantzas. An efficient component model for the construction of adaptive middleware. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, Middleware '01, pages 160–178, London, UK, UK, 2001. Springer-Verlag.

[Cointe, 1987] Pierre Cointe. Metaclasses are first class: The objvlisp model. *SIGPLAN Not.*, 22(12):156–162, Décembre 1987.

[Costa *et al.*, 2006] Fábio Moreira Costa, Lucas Luiz Provensi, et Frederico Forzani Vaz. Using runtime models to unify and structure the handling of meta-information in reflective middleware. In

Proceedings of the 2006 international conference on Models in software engineering, MoDELS'06, pages 232–241, Berlin, Heidelberg, 2006. Springer-Verlag.

[Costa Soria, 2011] Cristóbal Costa Soria. *Dynamic Evolution and Reconfiguration of Software Architectures through Aspects*. PhD thesis, Universidad Politécnica de Valencia, Spain, Juin 2011.

[Crnkovic *et al.*, 2011] I. Crnkovic, S. Sentilles, A. Vulgarakis, et M.R.V. Chaudron. A classification framework for software component models. *Software Engineering, IEEE Transactions on*, 37(5):593 –615, sept.-oct. 2011.

[Cuesta *et al.*, 2002] Carlos E. Cuesta, Pablo Fuente, Manuel Barrio-Solórzano, et M.Encarnación Beato. Introducing reflection in architecture description languages. In *Software Architecture*, éditeurs Jan Bosch, Morven Gentleman, Christine Hofmeister, et Juha Kuusela, volume 97 de *IFIP — The International Federation for Information Processing*, pages 143–156. Springer US, 2002.

[Curry *et al.*, 1982] Gael Curry, Larry Baer, Daniel Lipkie, et Bruce Lee. Traits: An approach to multiple-inheritance subclassing. *ACM SIGOA Newsletter*, 3(1-2):1–9, Juin 1982.

[Dashofy *et al.*, 2001] E. M. Dashofy, A. van der Hoek, et R. N. Taylor. A highly-extensible, XML-based architecture description language. In *Software Architecture, 2001. Proceedings. Working IEEE/IFIP Conference on*, pages 103–112, 2001.

[Daubert *et al.*, 2012] E. Daubert, F. Fouquet, O. Barais, G. Nain, G. Sunye, J.-M. Jezequel, J-L Pazat, et B. Morin. A models@runtime framework for designing and managing service-based applications. In *Software Services and Systems Research - Results and Challenges (S-Cube), 2012 Workshop on European*, pages 10–11, 2012.

[de Alfaro et Henzinger, 2001] Luca de Alfaro et Thomas A. Henzinger. Interface automata. In *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 109–120, New York, NY, USA, 2001. ACM Press.

[Demers et Malenfant, 1995] François-Nicola Demers et Jacques Malenfant. Reflection in logic, functional and object-oriented programming: a short comparative study. In *In IJCAI '95 Workshop on Reflection and Metalevel Architectures and their Applications in AI*, pages 29–38, 1995.

[Desnos *et al.*, 2007] Nicolas Desnos, Marianne Huchard, Christelle Urtado, Sylvain Vauttier, et Guy Tremblay. Automated and unanticipated flexible component substitution. In *Proceedings of the 10th ACM SIGSOFT Symposium on Component-Based Software Engineering (CBSE2007)*, éditeurs H. W. Schmidt et al., volume 4608 de *LNCS*, pages 33–48, Medford, MA, USA, July 2007. Springer.

[Dony *et al.*, 1992] Christophe Dony, Jacques Malenfant, et Pierre Cointe. Prototype-Based Languages: From a New Taxonomy to Constructive Proposals and Their Validation. In *OOPSLA*, pages 201–217, 1992.

[Ducasse *et al.*, 2006] Stéphane Ducasse, Tudor Gîrba, et Adrian Kuhn. Distribution Map. In *Proceedings International Conference on Software Maintenance (ICSM 2006)*, pages 203–212, Los Alamitos CA, 2006. IEEE Computer Society.

[Ducasse et Gîrba, 2006] Stéphane Ducasse et Tudor Gîrba. Using smalltalk as a reflective executable meta-language. In *Proceedings of the 9th international conference on Model Driven Engineering Languages and Systems*, MoDELS'06, pages 604–618, Berlin, Heidelberg, 2006. Springer-Verlag.

[Ducournau, 2002] Roland Ducournau. "real world" as an argument for covariant specialization in programming and modeling. In *Advances in Object-Oriented Information Systems*, Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2002.

[Ducournau, 2011] Roland Ducournau. Implementing statically typed object-oriented programming languages. Rapport technique, Montpellier II University, France, 2011.

[E. Chailloux et B. Pagano, 2004] P. Manoury E. Chailloux et O'Reilley B. Pagano. Développement d'applications avec Objective CAML. *Journal of Functional Programming*, 14(5):592–594, 2004.

[Emmerich, 2002] Wolfgang Emmerich. Distributed component technologies and their software engineering implications. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 537–546, New York, NY, USA, 2002. ACM.

[Fabresse *et al.*, 2008] Luc Fabresse, Christophe Dony, et Marianne Huchard. Foundations of a simple and unified component-oriented language. *Comput. Lang. Syst. Struct.*, July 2008.

[Fabresse *et al.*, 2012] Luc Fabresse, Noury Bouraqadi, Christophe Dony, et Marianne Huchard. A language to bridge the gap between component-based design and implementation. *COMLAN : Journal on Computer Languages, Systems and Structures*, 38(1):29–43, Avril 2012.

[Fabresse, 2007] Luc Fabresse. *From decoupling to unanticipated assembly of components: design and implementation of the component-oriented language Scl.* PhD thesis, Montpellier II University, Montpellier, France, December 2007.

[Flanagan, 1998] David Flanagan. *JavaScript: The Definitive Guide.* O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1998.

[Flatt, 2000] Matthew Raymond Flatt. *Programming languages for reusable software components.* PhD thesis, 2000. Adviser-Matthias Felleisen.

[Fowler, 2001] M. Fowler. To be explicit [software design]. *Software, IEEE*, 18(6):10–15, 2001.

[Gamma *et al.*, 1995a] Erich Gamma, Richard Helm, Ralph Johnson, et John Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software.* Addison Wesley, March 1995.

[Gamma *et al.*, 1995b] Erich Gamma, Richard Helm, Ralph E. Johnson, et John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, MA, 1995.

[Garlan *et al.*, 1994] David Garlan, Robert Allen, et John Ockerbloom. Exploiting style in architectural design environments. In *Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering*, SIGSOFT '94, pages 175–188, New York, NY, USA, 1994. ACM.

[Garlan *et al.*, 1997] David Garlan, Robert Monroe, et David Wile. Acme: an architecture description interchange language. In *Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research*, CASCON '97, pages 7–. IBM Press, 1997.

[Garlan et Kompanek, 2000] David Garlan et Andrew J. Kompanek. Reconciling the needs of architectural description with object-modeling notations. In *Proceedings of the 3rd international conference on The unified modeling language: advancing the standard*, pages 498–512, Berlin, Heidelberg, 2000. Springer-Verlag.

[Goldberg et Robson, 1989] Adele Goldberg et David Robson. *Smalltalk-80: The Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989.

[GoPivotal, Inc., 2013] GoPivotal, Inc. *Spring Framework Reference Documentation*. GoPivotal, Inc., 2013.

[Group, 1997] The Open Group. *Remote Procedure Call*. The Open Group, document number: c706 édition, 1997.

[Hamilton, 1997] Graham Hamilton. JavaBeans. API Specification, Sun Microsystems, Juillet 1997. Version 1.01.

[Henning et Vinoski, 1999] Michi Henning et Steve Vinoski. *Advanced CORBA programming with C++*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[Hnetynka et Pise, 2004] Petr Hnetynka et Michal Pise. Hand-written vs. mof-based metadata repositories: The sofa experience. In *ECBS*, pages 329–336. IEEE Computer Society, 2004.

[Hnětynka et Plášil, 2006] Petr Hnětynka et František Plášil. Dynamic reconfiguration and access to services in hierarchical component models. In *Proceedings of the 9th international conference on Component-Based Software Engineering*, CBSE'06, pages 352–359, Berlin, Heidelberg, 2006. Springer-Verlag.

[Hoare, 1978] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, Aôut 1978.

[Hunt et Thomas, 1999] Andrew Hunt et David Thomas. *The pragmatic programmer: from journeyman to master*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[Hürsch, 1994] Walter L. Hürsch. Should Superclass be Abstract? In *Proceedings ECOOP'94 : 8th European Conf. Object-Oriented Programming*, éditeurs M. Tokoro et R.Pareschi, volume 821 de *LNCS*, pages 12–31. Springer Verlag, July 1994.

[Ingalls, 1981] Daniel H. H. Ingalls. Design principles behind smalltalk. *BYTE Magazine*, August 1981.

[Kiczales *et al.*, 2001] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, et William G. Griswold. An Overview of AspectJ. In *ECOOP*, éditeur Jørgen Lindskov Knudsen, volume 2072 de *Lecture Notes in Computer Science*, pages 327–353. Springer, 2001.

[Kon *et al.*, 2000] Fabio Kon, Manuel Román, Ping Liu, Jina Mao, Tomonori Yamane, Claudio Magalhã, et Roy H. Campbell. Monitoring, security, and dynamic configuration with the dynamictao reflective orb. In *IFIP/ACM International Conference on Distributed systems platforms*, Middleware '00, pages 121–143, Secaucus, NJ, USA, 2000. Springer-Verlag New York, Inc.

[Kriens, 2012] Peter Kriens. Simplexity, the quest for reuse. Key talk in Conférence sur les Architectures Logicielles (CAL), Montpellier, France, 2012.

[Kruchten *et al.*, 2009] P. Kruchten, R. Capilla, et J.C. Dueas. The decision view's role in software architecture practice. *Software, IEEE*, 26(2):36–42, 2009.

[Lahire *et al.*, 2004] P. Lahire, G. Arévalo, H. Astudillo, A.P. Black, E. Ernst, M. Huchard, T. Oplustil, M. Sakkinen, et P. Valtchev. Mechanisms for Specialization, Generalization and Inheritance, 2004. MASPEGHI.

[Lahire et Quintian, 2006] Philippe Lahire et Laurent Quintian. New Perspective To Improve Reusability in Object-Oriented Languages. *Journal Of Object Technology (JOT)*, 5(1):117–138, 2006.

[Langelier *et al.*, 2005] Guillaume Langelier, Houari A. Sahraoui, et Pierre Poulin. Visualisation et analyse de logiciels de grande taille. In *Langages et Modèles à Objets 2005*, Mars 2005.

[Lau et Wang, 2005a] K. K. Lau et Z. Wang. A Survey of Software Component Models. Technical reports, Department of Computer Science, University of Manchester, April 2005.

[Lau et Wang, 2005b] Kung-Kiu Lau et Zheng Wang. A taxonomy of software component models. In *EUROMICRO-SEAA*, pages 88–95. IEEE Computer Society, 2005.

[Leclercq *et al.*, 2007] Matthieu Leclercq, Ali Erdem Ozcan, Vivien Quema, et Jean-Bernard Stefani. Supporting Heterogeneous Architecture Descriptions in an Extensible Toolset. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 209–219, Washington, DC, USA, 2007. IEEE Computer Society.

[Ledoux et Cointe, 1996] Thomas Ledoux et Pierre Cointe. Explicit metaclasses as a tool for improving the design of class libraries. In *Proceedings of the Second JSSST International Symposium on Object Technologies for Advanced Software*, ISOTAS '96, pages 38–55, London, UK, UK, 1996. Springer-Verlag.

[Ledoux, 1999] Thomas Ledoux. Opencorba: A reflective open broker. In *Proceedings of the Second International Conference on Meta-Level Architectures and Reflection*, Reflection '99, pages 197–214, London, UK, UK, 1999. Springer-Verlag.

[Lieberman, 1986a] Henry Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In *OOPLSA '86: Conference proceedings on object-oriented programming systems, languages and applications*, pages 214–223, New York, NY, USA, 1986. ACM Press.

[Lieberman, 1986b] Henry Lieberman. Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems. In *procs. of OOPSLA*, Portland, Oregon, USA, Novembre 1986. Published as ACM SIGPLAN Notices 21(11).

[Lippman, 1996] Stanley B. Lippman. *Inside the C++ object model*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1996.

[Liskov et Zilles, 1974] Barbara Liskov et Stephen Zilles. Programming with abstract data types. In *Proceedings of the ACM SIGPLAN symposium on Very high level languages*, New York, NY, USA, 1974. ACM.

[Luckham *et al.*, 1995a] David C. Luckham, John J. Kenney, Larry M. Augustin, James Vera, Doug Bryan, et Walter Mann. Specification and analysis of system architecture using rapide. *IEEE Trans. Softw. Eng.*, 21:336–355, April 1995.

[Luckham *et al.*, 1995b] David C. Luckham, James Vera, et Sigurd Meldal. Three Concepts of System Architecture. Rapport Technique CSL-TR-95-674, 1995.

[Luckham et Vera, 1995] David C. Luckham et James Vera. An event-based architecture definition language. *IEEE Trans. Softw. Eng.*, 21(9):717–734, Septembre 1995.

[Léger *et al.*, 2006] Marc Léger, T. Coupaye, et Thomas Ledoux. Contrôle dynamique de l'intégrité des communications dans les architectures à composants. In *Langages et Modèles à Objets*, éditeur S. Vauttier R. Rousseau, C. Urtado, pages 21–36. Hermès-Lavoisier, 2006.

[Maes, 1987] Pattie Maes. Concepts and experiments in computational reflection. *SIGPLAN Not.*, 22(12):147–155, Décembre 1987.

[Magee *et al.*, 1995] Jeff Magee, Naranker Dulay, Susan Eisenbach, et Jeff Kramer. Specifying distributed software architectures. In *Proceedings of the 5th European Software Engineering Conference*, pages 137–153, London, UK, 1995. Springer-Verlag.

[Malenfant *et al.*, 1992] J. Malenfant, C. Dony, et P. Cointe. Behavioral reflection in a prototype-based language. In *Proceedings of International Workshop on Reflection and Meta-Level Architectures*, pages 143–153. ACM, 1992.

[Malenfant *et al.*, 1996] Jacques Malenfant, M. Jacques, et François-Nicola Demers. A tutorial on behavioural reflection and its implementation. In *Proceedings of the First International Conference on Reflection, Reflection'96*, Reflection '96, pages 1–20, 1996.

[Martin, 2002] Robert C. Martin. *Agile Software Development, Principles, Patterns, and Practices*. Prentice-Hall, Inc, 2002.

[McIlroy, 1968] M. D. McIlroy. Mass produced software components. In *Proceedings, NATO Conference on Software Engineering*, éditeurs P. Naur et B. Randell, Garmisch, Germany, Octobre 1968.

[McIlroy, 1972] M. D. McIlroy. The outlook for software components. In *Software Engineering*, pages 243–252. Infotech Information, Ltd., Maidenhead, England, 1972.

[McVeigh *et al.*, 2006] Andrew McVeigh, Jeff Kramer, et Jeff Magee. Using resemblance to support component reuse and evolution. In *Procs. of SAVCBS*, New York, NY, USA, 2006. ACM.

[Medvidovic *et al.*, 1996] Nenad Medvidovic, Peyman Oreizy, Jason E. Robbins, et Richard N. Taylor. Using object-oriented typing to support architectural design in the c2 style. *SIGSOFT Softw. Eng. Notes*, 21(6):24–32, Octobre 1996.

[Medvidovic *et al.*, 1997] Nenad Medvidovic, Peyman Oreizy, et Richard N. Taylor. Reuse of off-the-shelf components in c2-style architectures. In *Proceedings of the 1997 symposium on Software reusability*, SSR '97, pages 190–198, New York, NY, USA, 1997. ACM.

[Medvidovic et Taylor, 2000] Nenad Medvidovic et Richard N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *Software Engineering*, 26(1):70–93, 2000.

[Mehta *et al.*, 2000] Nikunj R. Mehta, Nenad Medvidovic, et Sandeep Phadke. Towards a taxonomy of software connectors. In *Proceedings of the 22nd international conference on Software engineering*, ICSE '00, pages 178–187, New York, NY, USA, 2000. ACM.

[Mens, 2008] Tom Mens. Introduction and roadmap: History and challenges of software evolution. In *Software Evolution*, pages 1–11. Springer Berlin Heidelberg, 2008.

[Mernik *et al.*, 2005] Marjan Mernik, Jan Heering, et Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Computer Surveys*, 37(4):316–344, 2005.

[Meyer, 2001] Bertrand Meyer. Overloading vs object technology. *Journal of Object-Oriented Programming (JOOP)*, 14(4):3–7, October-November 2001.

[Microsoft, 1995] Microsoft. The Component Object Model Specification, 1995.

[Microsoft, 2012] Microsoft. *COM: Component Object Model Technologies*. Microsoft, 2012.

[Mikhajlov et Sekerinski, 1998] Leonid Mikhajlov et Emil Sekerinski. A Study of the Fragile Base Class Problem. *Lecture Notes in Computer Science*, 1445:355, 1998.

[Monroe, 2001] Robert T. Monroe. Capturing software architecture design expertise with armani. Rapport technique, School of Computer Science, Carnegie Mellon University,Pittsbugh, Pennsylvania, USA, 2001.

[Monson-Haefel, 1999] Richard Monson-Haefel. *Enterprise JavaBeans*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1999.

[Morin *et al.*, 2009] Brice Morin, Olivier Barais, Gregory Nain, et Jean-Marc Jezequel. Taming dynamically adaptive systems using models and aspects. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 122–132, Washington, DC, USA, 2009. IEEE Computer Society.

[Muller *et al.*, 2005] Pierre-Alain Muller, Franck Fleurey, et Jean-Marc Jezequel. Weaving executability into object-oriented meta-languages. In *Proceedings of the 8th international conference on Model Driven Engineering Languages and Systems*, MoDELS'05, Berlin, Heidelberg, 2005. Springer-Verlag.

[Nierstrasz *et al.*, 2005] Oscar Nierstrasz, Alexandre Bergel, Marcus Denker, Stéphane Ducasse, Markus Gälli, et Roel Wuyts. On the revival of dynamic languages. In *Proceedings of Software Composition 2005*, éditeurs Thomas Gschwind et Uwe Aßmann, volume 3628, pages 1–13. LNCS 3628, 2005.

[OASIS, 2013] OASIS. *Service Component Architecture (SCA)*. Organization for the Advancement of Structured Information Standards, 2013.

[Odersky et Zenger, 2005] Martin Odersky et Matthias Zenger. Scalable Component Abstractions. In *OOPSLA*, éditeurs Ralph Johnson et Richard P. Gabriel, pages 41–57. ACM, 2005.

[OMG, 2011a] OMG. *Meta Object Facility (MOF) Core Specification Version 2.4.1*, 2011.

[OMG, 2011b] OMG. UML 2.4.1 superstructure specification; document formal/2011-08-06. Rapport technique, OMG, August 2011.

[OMG, 2011c] OMG. *Unified Modeling Language (UML), V2.4.1*. OMG, August 2011.

[OMG, 2012] OMG. *CORBA Component Model (CCM)*. OMG, 2012.

[Oplustil, 2002] T. Oplustil. Inheritance of sofa components. Master's thesis, Faculty of Informatics, Masaryk University, Brno, Czech Republic, June 2002.

[Opluštil, 2003] Tomáš Opluštil. Inheritance in Architecture Description Languages. In *WDS 2003 - Proceedings of Contributed Papers*, éditeur Jana Šafránková, pages 124–131, Prague, Czech Republic, 2003. Matfyzpress, MFF UK.

[Oracle, 2012] Oracle. *Enterprise JavaBeans Specification Version 3*. Oracle, 2012.

[OSGi Alliance, 2012] OSGi Alliance. *OSGi Core Release 5 Specification*. OSGi Alliance, 2012.

[Oussalah *et al.*, 2006] M. Oussalah, N. Sadou, et D. Tamzalit. SAEV :A Model to Face Evolution Problem in Software Architecture. In *Proceedings of the International ERCIM Workshop on Software Evolution*, pages 137–146, Lille, France, April 2006.

[Outhred et Potter, 1998] G Outhred et J Potter. A model for component composition with sharing. In *Proceedings of the Workshop on Component Oriented Programming (WCOP)*. ECOOP Workshop Reader, 1998.

[Pavel *et al.*, 2005] Sebastian Pavel, Jacques Noyé, et Jean-Claude Royer. Un modèle de composant avec protocole symbolique. In *Journée du groupe Objets, Composants et Modèles*, Bern, Suisse, 2005.

[Peschanski *et al.*, 2000] F. Peschanski, T. Meurisse, et J.-P. Briot. Les composants logiciels : Evolution technologique ou nouveau paradigme ? In *In Actes de la conférence OCM'2000*, pages 53–65, 2000.

[Plásil *et al.*, 1998] F. Plásil, D. Bálek, et R. Janecek. SOFA/DCUP: Architecture for component trading and dynamic updating. In *CDS '98: Proceedings of the International Conference on Configurable Distributed Systems*, page 43, Washington, DC, USA, 1998. IEEE Computer Society.

[Plásil *et al.*, 1999] Frantisek Plásil, Miloslav Besta, et Stanislav Visnovsky. Bounding component behavior via protocols. In *TOOLS '99: Proceedings of the Technology of Object-Oriented Languages and Systems*, page 387, Washington, DC, USA, 1999. IEEE Computer Society.

[Privat, 2006] Jean Privat. *De l'expressivité à l'efficacité, une approche modulaire des langages à objets. Le langage PRM et le compilateur* `prmc`. PhD thesis, Université de Montpellier 2, LIRMM, Juillet 2006.

[Provensi *et al.*, 2010] Lucas Luiz Provensi, Fábio Moreira Costa, et Vagner Sacramento. Management of runtime models and meta-models in the meta-orb reflective middleware architecture, 2010.

[Renggli *et al.*, 2010] Lukas Renggli, Stéphane Ducasse, Tudor Gîrba, et Oscar Nierstrasz. Practical Dynamic Grammars for Dynamic Languages. In *4th Workshop on Dynamic Languages and Applications (DYLA 2010)*, Malaga, Espagne, 2010.

[Rivard, 1996] F Rivard. A new smalltalk kernel allow ing both explicit and implicit metaclass programming. In *Workshop on Extending the Smalltalk Language*, OOPSLA '96, October 1996.

[Rogerson, 1997] Dale Rogerson. *Inside COM*. Microsoft Press, Redmond, WA, USA, 1997.

[Sánchez Cuadrado, 2012] Jesús Sánchez Cuadrado. Towards a family of model transformation languages. In *Proceedings of the 5th international conference on Theory and Practice of Model Transformations*, ICMT'12, pages 176–191, Berlin, Heidelberg, 2012. Springer-Verlag.

[Schmid et Pfeifer, 2008] Hans Albrecht Schmid et Marco Pfeifer. Engineering a component language: Compjava. In *Software and Data Technologies*, Communications in Computer and Information Science. Springer Berlin Heidelberg, 2008.

[Seco *et al.*, 2008] J. C. Seco, Ricardo Silva, et Margarida Piriquito. Componentj: A component-based programming language with dynamic reconfiguration. *Computer Science and Information Systems*, 05(02):65–86, 12 2008.

[Seco et Caires, 2000] João Costa Seco et Luís Caires. A basic model of typed components. *Lecture Notes in Computer Science*, 1850:108–129, 2000.

[Seinturier *et al.*, 2006] L. Seinturier, N. Pessemier, L. Duchien, et T. Coupaye. A model for developing component-based and aspect-oriented systems. In *Proceedings of the 5th International Symposium on Software Composition (SC'06)*, volume 4089 de *Lecture Notes in Computer Science*. Springer, Mars 2006.

[Seinturier *et al.*, 2012] Lionel Seinturier, Philippe Merle, Romain Rouvoy, Daniel Romero, Valerio Schiavoni, et Jean-Bernard Stefani. A component-based middleware platform for reconfigurable service-oriented architectures. *Softw. Pract. Exper.*, 42(5):559–583, Mai 2012.

[Shaw *et al.*, 1995] Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, et Gregory Zelesnik. Abstractions for software architecture and tools to support them. *IEEE Trans. Softw. Eng.*, 21(4):314–335, Avril 1995.

[Shaw et Garland, 1996] Mary Shaw et David Garland. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, Upper Saddle River, NJ, 1996.

[Smith, 1982] Brian Cantwell Smith. *Procedural Reflection in Programming Languages*. PhD thesis, Massachusetts Institute of Technology, Laboratory for Computer Science, 1982.

[Snyder, 1987] Alan Snyder. Inheritance and the development of encapsulated software systems. In *Research Directions in Object-Oriented Programming*, pages 165–188. 1987.

[Souchon, 2005] Frédéric Souchon. *SaGE, Un Système de Gestion d'Exceptions pour la Programmation Orientée Message : Le Cas des Systèmes Multi-Agents et des Plates-Formes à Base de Composants Logiciels*. PhD thesis, Université de Montpellier 2, LIRMM, 2005.

[Spacek *et al.*, 2012] Petr Spacek, Christophe Dony, Chouki Tibermacine, et Luc Fabresse. An inheritance system for structural & behavioral reuse in component-based software programming. In *Proceedings of the 11th GPCE*, pages 60–69. ACM, 2012.

[Sreedhar, 2002] Vugranam C. Sreedhar. Mixin'Up components. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 198–207, New York, NY, USA, 2002. ACM Press.

[Stein, 1987] Lynn Andrea Stein. Delegation is inheritance. In *OOPSLA '87: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 138–146, New York, NY, USA, 1987. ACM Press.

[Stepanov et Lee, 1994] Alexander Stepanov et Meng Lee. The Standard Template Library. Rapport technique, ISO Programming Language C++ Project, 1994.

[Szyperski, 2002] C. Szyperski. *Component Software: Beyond Object-Oriented Programming (2nd Edition)*. Addison-Wesley, 2002.

[Taenzer *et al.*, 1989] D. Taenzer, M. Ganti, et S. Podar. Problems in Object-Oriented Software Reuse. In *Proceedings of ECOOP'89 : European Conf. Object-Oriented Programming*, éditeur S. Cook, volume 821, pages 25–38. Cambridge University Press, july 1989.

[Tang *et al.*, 2005] A. Tang, M.A. Babar, I. Gorton, et J. Han. A survey of the use and documentation of architecture design rationale. In *Software Architecture, 2005. WICSA 2005. 5th Working IEEE/IFIP Conference on*, pages 89–98, 2005.

[Terrier et Gérard, 2006] François Terrier et Sébastien Gérard. MDE Benefits for Distributed, Real Time and Embedded Systems. In *From Model-Driven Design to Resource Management for Distributed Embedded Systems*, volume 225 de *IFIP International Federation for Information Processing*, chapitre 3, pages 15–24. Springer Boston, Boston, MA, 2006.

[Tibermacine *et al.*, 2010a] Chouki Tibermacine, Christophe Dony, Salah Sadou, et Luc Fabresse. Software architecture constraints as customizable, reusable and composable entities. In *Proceedings of the 4th European Conference on Software Architecture (ECSA'10)*, Copenhagen, Denmark, August 2010. Springer-Verlag.

[Tibermacine *et al.*, 2010b] Chouki Tibermacine, Régis Fleurquin, et Salah Sadou. A family of languages for architecture constraint specification. *In the Journal of Systems and Software (JSS), Elsevier*, 2010.

[Tibermacine *et al.*, 2011] Chouki Tibermacine, Salah Sadou, Christophe Dony, et Luc Fabresse. Component-based specification of software architecture constraints. In *Proceedings of the 14th CBSE*, pages 31–40, New York, NY, USA, 2011. ACM.

[Tremblay et Chae, 2005] Guy Tremblay et Junghyun Chae. Towards specifying contracts and protocols for Web services. In *MCeTech Montreal Conference on eTechnologies*, éditeurs H. Mili et F. Khendek, pages 73–85, January 2005.

[Ungar et Smith, 1987] David Ungar et Randall B. Smith. Self: The power of simplicity. In *OOPSLA '87: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 227–242, New York, NY, USA, 1987. ACM Press.

[Vraný *et al.*, 2012] Jan Vraný, Jan Kurs, et Claus Gittinger. Efficient method lookup customization for smalltalk. In *TOOLS (50)*, pages 124–139, 2012.

[Weck et Szyperski, 1996] W. Weck et C. Szyperski. Do we need inheritance. In *CIOO Workshop at ECOOP, Linz*, December 1996.

[Wettel et Lanza, 2007] Richard Wettel et Michele Lanza. Visualizing Software Systems as Cities. In *VISSOFT'07 (4th IEEE International Workshop on Visualizing Software For Understanding and Analysis)*, éditeur IEEE CS Press, pages 92–99, 2007.

[Wuyts et Ducasse, 2001] Roel Wuyts et Stéphane Ducasse. Composition Languages for Black-Box Components. In *First OOPSLA Workshop on Language Mechanisms for Programming Software Components*, 2001.

[Xu et Ren, 2010] Liping Xu et Yufei Ren. Bichon: A new component-oriented programming language. *Software Engineering, World Congress on*, 2010.

[Zenger, 2002] Matthias Zenger. Type-safe prototype-based component evolution. In *Proceedings of the European Conference on Object-Oriented Programming*, Malaga, Spain, June 2002.