

ARCHITECTURAL DESCRIPTION OF OBJECT ORIENTED FRAMEWORKS

Gabriela B. Arevalo

**Thesis Advisor: Isabelle Borne
(Ecole des Mines de Nantes)**

August 2000

Abstract

Nowadays, the existing formalisms to represent software architectures (such as box and line drawings) fail in providing a clear semantics and only give an intuitive picture of the system as a whole, which is not enough as a valuable description. More specifically, the framework architectures should show the overall design and the specification of the points of variability of the framework, making easier the reuse of the architecture, integration with other frameworks and a reference to measure the changes in subsequent versions of the frameworks. Starting from several frameworks, we propose, as a first step, to study and compare the various levels and expressive power of two formal approaches, such as architectural patterns and Wright –an architectural description language. Next we study the possible complementarity of these approaches, and also evaluate the flexibility of the descriptions in order to be able to take evolution aspects into account. The final objective is to propose a complete description of a framework based on the previous results.

Acknowledgements

After all this year, I've got quite a list of people who contributed in some way to this thesis, for which I would like to express thanks.

To Isabelle Borne, my thesis advisor, who gave me guidance and support throughout the entire thesis process. Thank you !

To Gustavo Rossi, who involved me in this international program. He also helped me through many problems, and his opinions helped shape this work. Thank you !

To Annya and Janick, who worked very hard to provide me a good environment to study and to live in Nantes. Thank you !

To Remi Douence, who shared with me all his knowledge about softwares architectures. Thank you !

To Xavier Alvarez, who helped me with his fruitful discussions to focus the goals of this thesis. Thank you !

A mis compañeros del LIFIA : Mauricio, Fernando y Ramiro por ayudarme al principio de esta tesis cuando mis conocimientos sobre frameworks era basico y ellos respondieron con mucha paciencia a todas mis preguntas. Thank you !

A mis amigos en Nantes : Sinagi, Andres, Gabriel y Xavier por convertirse en mi familia durante mi estadia en Nantes. Thank you !

A mis « hermanos postizos » : Guillermo, Anabella, Alejandra y Cristian (en Argentina) y Alejandro y Analia (en Alemania) por bancarme en los buenos y en los malos momentos. Sus mails, cartas, postales y la bandera argentina que recibí de regalo hicieron que no sintiera la distancia fisica con ellos y que el cariño que les tengo crezca día a día. Thank you !

A mis mejores amigas Karina y Nora por su cariño y quienes estuvieron siempre presentes con su corazon al lado mio. Thank you !

A mis hermanos Dario, Patricia, Cynthia, Virna y Victor por sus chistes, sus anecdotas y su alegría que transmitieron por e-mail haciendome sentir como si yo estuviera entre ellos en mi casa. Thank you !

Especialmente a Papá y Mamá por apoyarme en este nuevo proyecto de mi vida y alentarme diariamente para que no bajara los brazos ante las adversidades tan sencillas como la de estar lejos fisicamente de mi casa y de Argentina. Sin esa fuerza y ese amor inconmensurable que transmiten en el día a día, esta experiencia hubiera sido imposible. A ellos les dedico mi trabajo.

Gabriela Arévalo

August 2000

Table of Contents

Chapter 1: Introduction	13
1.1 Motivation	13
1.2 Current Research	14
1.2.1 Documenting Frameworks using Patterns [Joh92]	14
1.2.2 Design Patterns, Contracts, and Motifs in Concert [LK94]	15
1.2.3 More than design patterns [Ric98]	16
1.3 Architectural Description Language as an alternative formal approach	17
1.4 Related Work	17
1.5 Approaches and Contribution	18
1.6 Organization of the Thesis	19
Chapter 2: Software Architectures and Approaches of Description	21
2.1 Definitions of Software Architecture	21
2.2 Architectural Patterns	22
2.3 Architectural Styles	24
2.3.1 Practical Benefits of Architectural Styles	26
2.4 Architectural Styles and Patterns	26
2.5 Significance of Software Architecture to Software Engineering	27
2.6 Architecture Description Languages (ADLs): General features	27
2.6.1 Architectural Elements and Its Properties	28
2.6.2 Wright: An Architectural Description Language	30
2.6.2.1 Structure	30
2.6.2.2 Style	32
2.6.2.3 Behaviour: Definition of interaction protocols	33
2.6.2.4 Validation of Descriptions	35
2.6.2.5 Properties	36
2.7 Summary	36
Chapter 3: Object Oriented Frameworks	37
3.1 Definition	37
3.2 Differences with other concepts	37
3.3 Characterization of Frameworks by different dimensions	38
3.4 Application Frameworks: Specific Features	39
3.5 Components and Frameworks	40
3.6 Keys of Frameworks	41
3.6.1 : Analysis, Design and Code Reuse	41
3.6.2 Hotspots and Frozen Spots	41
3.7 Analysis: Goals, Benefits and Weaknesses	42
3.7.1 Goals	42
3.7.2 Benefits	42
3.7.3 Weaknesses	43
3.8 Summary	43
Chapter 4: Architectural Description of Frameworks : First Problems	45
4.1 Software Architecture: Level of Abstraction	45
4.2 Architectural Description: Conceptual Model	46
4.3 Architectural Components = Software Components ?	48
4.4 Connectors	49
4.5 Internal State of the Components	50
4.5.1 Wright and Abstract Machines B	50
4.5.2 Interaction of the Abstract Machine B with the environment	51

4.5.3	Connection between a Wright Component and an Abstract Machine B	52
4.5.4	Description of the interaction between a Wright component and an Abstract Machine B	53
4.5.4.1	Resolution of non-determinism in the connector B	55
4.6	Summary	56
<u>Chapter 5: Architectural Description OF Object Oriented Frameworks: An Approach</u>		57
5.1	Introduction	57
5.2	MAPPINGS: Definition and Assumptions	59
5.2.1	Mapping for Classes	59
5.2.2	Mapping for Relationships between Classes	60
5.2.3	Format for Components and Connectors	61
5.2.4	Mapping for the Messages	61
5.2.5	Mapping for Classes' Creation	62
5.2.6	Mapping for Conditional Statements	63
5.3	Object oriented Architectural Description - First Case: Design and Code	64
5.3.1	Example: Measurements System Framework [Bos00]	64
5.3.1.1	Architecture of the System	65
5.3.1.2	Minimal details about the design	65
5.3.2	First Approach: Use of Predefined Styles	66
5.3.3	Second Approach: Use of Steps	67
5.4	Architectural Description – Second Case: Architectural Patterns + Wright Description	73
5.4.1	Example: A Generic Coordination Abstraction for Managing Shared Resources [CTN97]	73
5.4.2	Requirements	74
5.4.3	Solution	74
5.4.4	Approach: Use of Steps	75
5.5	Summary	82
<u>Chapter 6: Architectural Description Aspects: Discussion</u>		84
6.1	Analogy between Architectural Style and Framework Architecture	84
6.2	Levels of Description	87
6.3	Level of Components	88
6.4	The Potential of the Connectors	88
6.5	Increment of Connectors and Components	89
6.6	Management of Errors	89
6.7	Lack of Expressiveness in Structural Features	89
6.8	Lack of Expressiveness of Timing	89
6.9	Dynamic Binding and Creation/Destruction of Components	89
6.10	Detection of Errors with the Wright Checking Tool	90
6.11	Use and Discovering of Patterns in the Architectural Description	90
6.12	Properties of Cohesion and Coupling in the Architectural Description	91
6.13	Architectural Views	92
6.14	Complementaries with other Techniques	93
6.15	Summary	94
<u>Chapter 7: Evolution of Frameworks</u>		95
7.1	General Overview of the Levels of Abstraction	95
7.2	Aspects of Evolution	96
7.2.1	Internal reorganization	96
7.2.2	Changing functionality	98

7.2.3	Extending functionality	99
7.2.4	Reducing functionality	99
7.3	Analysis	100
7.4	Summary	100
Chapter 8: Conclusions and Future Research		101
8.1	Context	101
8.2	Conclusions	102
8.3	Future Research	103

List of Figures

Figure 2.1: Architectural Styles in graphic notation	24
Figure 2.2: Multi-Phase Architectural Style [PW92]	25
Figure 2.3: Hierarchical Structure	31
Figure 4.1: Conceptual Modelling of Software Architectures [Now99]	47
Figure 4.2: Component or Connector ? [SG99]	49
Figure 4.3: Specification in B of Text_Filter	51
Figure 4.4: Ports Component and B	51
Figure 4.5: An Abstract Machine B with its interface	52
Figure 4.6: Two possible connections between a Wright component and an Abstract Machine B	52
Figure 5.1: Composite Pattern	58
Figure 5.2: Graphical Relationships between classes	58
Figure 5.3: Class Model for a Book and the Mapping with Connectors and Components	60
Figure 5.4: Class Model for a Truck and Representation with Components and Connectors	61
Figure 5.5: Architecture of the Measurement System Frameworks	65
Figure 5.6: Master / Slave Pattern	67
Figure 5.7: Class Model for the Shared Resource Access Policy	75
Figure 5.8: Example of Application of Layers Architectural Pattern	76
Figure 5.9: The developed algorithms seen graphically	81
Figure 6.1: Growth of the Architectural Description in X-Y axis	87
Figure 7.1: Software Evolution at Three Levels of Abstraction	96

Chapter 1: Introduction

1.1 Motivation

An object oriented framework is a kind of reusable software architecture comprising both design and code. More specifically, [Mat96] defines an object oriented framework as a generative architecture for maximum reuse represented as a collective set of abstract and concrete classes, encapsulating potential behaviour for subclassed specializations.

One critical issue for users and implementors of a framework is the documentation that explains what the framework provides and what is required to instantiate it correctly for some application. Typically, a framework is specified using a combination of informal and semi-formal documentation. On the informal side are guidelines and high-level descriptions of usage scenarios, tips and examples [SG99]. If an object oriented methodology such as UML [UML] was used to document the framework, there are class and collaboration diagrams as a description artefacts. These approaches tend to be informal and idiosyncratic, consisting of box-and-line diagrams that convey the essential system structure, together with the prose that explains the meaning of the symbols [MKMG96]. On the semi-formal side one usually finds a description of an application programmer's interface (API) that explains what kinds of services are provided by the framework. APIs are formal to the extent that they provide precise descriptions of those services -usually as a set of signatures, possibly annotated with informal pre and post-conditions [SG99].

Such documentation is clearly necessary. However, by itself it leaves many important questions unanswered -for component developers, system integrators, framework implementors, and proposers of new frameworks. For example, the framework API may specify the names and parameters of services provided by the infrastructure. However, it may not be clear what are the restrictions (if any) on the ordering of invocations of those services. Usage scenarios may help, but they only provide examples of selected interactions, requiring the reader to infer the general rule. Moreover, it may not be clear what facilities must be provided by the parts added to the framework, and which are optional. [SG99].

As with most forms of informal system documentation and specification, the situation could be greatly improved if one had a precise description as a formal specification of the framework. However, a number of critical issues arise immediately. What aspects of the framework should be modeled? How should that model be structured to best expose the architectural design? How should one model the parts of the framework to maintain traceability to the original documentation, and yet still improve clarity? How should one distinguish optional from required behaviour? [SG99] For object oriented frameworks what aspects of the object-oriented design should be exposed in the formal model? . The model should answer questions as the following ones: Which new classes must be provided for the framework? What classes should be used from the framework? Which operations must be called and which operations are often called in the framework? What classes must be subclassed? Which operations must be overridden and which operations are often overridden? [Mat96].

About describing framework architectures, [Joh92] denotes that "*nobody understands a framework until they have used it*". There are several reasons to have an **architectural description of a framework based on high-level interfaces and interactions**, and characterizing their **semantics in terms of protocols**:

- ⇒ **'Selling' the framework**: Prospective users of the framework want a general understanding of the framework in order to decide if it is appropriate for their needs. This kind of description should show the overall design and the points of variability of the framework. [Ric98].
- ⇒ **Reuse of architecture**: Transmitting a language-independent view of the architecture allows the high-level design of the framework to be reused in implementing it in other languages, or in modifying it for use in other domains. [Ric98]
- ⇒ **Integration of frameworks**: In order to facilitate the construction of systems from several existing frameworks, the architectural assumptions of each framework should be made explicit. [GAO95] [MB00]
- ⇒ **Evolution and re-engineering**: Having an architectural description of a framework gives us a reference against which one can measure the changes in subsequent versions of the framework. In the same way, the ability to describe the architecture of an application allows us to form hypothesis about the architecture which can be tested in the process of reverse engineering. [MN97]

This thesis aims to propose a complete description of a framework using the following formal approaches: architectural patterns and Wright [All97] -an architectural description language. The steps to fulfill this objective consist of studying and comparing the various levels and expressive power of approaches, also studying the possible complementarity and flexibility of these approaches in order to be able to take evolution aspects into account.

1.2 Current Research

There are no well-established rules for describing framework. Thus, different approaches have been proposed focusing mostly on informal techniques. We will enumerate some of them, where the ideas are similar to our approach: the use of several artifacts of domain analysis and object oriented design to give different views of a framework's architecture.

1.2.1 Documenting Frameworks using Patterns [Joh92]

This approach deals with the frameworks description using pattern language. A pattern language is a set of patterns, where each of them describes how to solve a particular kind of problem. The main idea is to start from a very large scale and going into further details from pattern to pattern. The approach is based on the fact that the framework users want to know as little as possible about the framework. Thus, patterns have to go through all functionality of the framework from a general presentation to more precise features and then progressively to describe which the hotspots and the collaboration contracts are. This approach fulfills its three purposes of the description of:

- *the purpose of the framework*: It must be the first thing that the documentation describes. If the framework turns out to be inappropriate then the reader does not have to continue reading. The first pattern describes its application domain. It acts as a entry catalog for the framework and a road map regarding the rest of patterns
- *how to use the framework*: The patterns give a kind of cookbook, giving detailed instructions for using the frameworks.
- *the detailed design of the framework*: This information not only includes the different classes in the framework but also the way that the instances of these classes

collaborate. The idea is to treat the set of patterns as a directed graph, using the references from one pattern to another as the edges, and to place design information as far from the first pattern as was feasible.

The use of examples in this approach makes frameworks more concrete, makes it easier to understand the flow of control, and helps the reader to determine whether he or she understands the rest of documentation.

Summarizing up, patterns are an informal technique aimed primarily at describing how to use a framework, not describing its algorithms, patterns of collaboration or shared invariant. They just describe specific points of the framework supported by examples.

1.2.2 Design Patterns, Contracts, and Motifs in Concert [LK94]

This approach develops a multi-layered model for framework reuse which comprises reuse objects at different levels of abstraction and varying degrees of encapsulation, most notably, microarchitectures. It integrates design patterns, contracts, and motifs as techniques for the representation and documentation of microarchitectures and frameworks. It also suggests a design guided by the design of the underlying framework and possibly of its subframeworks and that a mixed top-down/bottom-up approach be taken.

[GHJV95] detected that there were redundant design structures into the framework and introduced the concept of microarchitectures as a set of classes and its interactions. As frameworks codify design knowledge of a particular domain, microarchitectures codify knowledge in terms of the behavior of object collaborations. Microarchitectures comprise both the design and code of the classes involved and the interactions and control flow among them. These structures are of course known by the framework designers, but unfortunately by very few others. Design patterns are a mechanism to express how component interrelate as well as high-level representation technique for properly capturing and expressing design experience and intent to ultimately facilitate design reuse. They can be described in an informal, template-based manner. But the design patterns describe the framework design at a very high level. To ease the derivation of concrete designs from them, a contract - a construct for specifying interactions among groups of objects - is used as an intermediate representation between a microarchitecture and its corresponding design patterns. Contracts used to describe frameworks provide the application designer with:

- A vocabulary with which to describe the application;
- Through conformance declarations, the identification of the application-specific classes, variables, methods, and hooks for customization, all necessary for identifying, maintaining and implementing a behavioral composition;
- Knowledge of, and a better understanding of, individual microarchitectures present in the underlying framework, thus improving the understanding of the overall framework;
- Guidance when refactoring affects participants in a contract. Contracts should not be taken as a means to understand the functionality of classes and methods.

More oriented to novice users, the motifs show how to use the framework. Each of them describes a situation which must be replicated in order to use the framework. A set of motifs describe the underlying class framework and an example application.

To put all the techniques together, this approach covers a documentation with:

- The purpose of and how to use the underlying framework;
- The purpose of and how to reuse the application examples;
- The design of the framework

Motifs are responsible for describing the first two items, whereas design patterns and contracts address the last item. Thus motifs complement design documentation doing it so informally without detailing algorithms or object collaborations. Instead they often refer to design patterns and/or contracts. To support documentation from lower level of a framework, each class (and even if it is possible, also the individual methods) should have references to motifs, design patterns, and/or contracts it is involved in.

With this proposal, they discover that design patterns, contracts, and motifs, along with the corresponding code components, classes and methods, must cross reference each other whenever appropriate.

1.2.3 More than design patterns [Ric98]

This approach analyzes the insufficiency of design patterns to get an architectural description of a framework. Considering the selling issue, design patterns do not describe what we can do with the framework, how we can tailor the framework to create a variety of applications. Taking into account reuse issue, reusing a framework implies understanding the domain-specific architecture that the framework offers, knowing how it models the problem domain, and what the model components of the system and their relationships. Indeed the granularity of design patterns is generally very fine and is not always suited to the granularity of the main components of a framework. For example, behavioral patterns ([GHJV95]) are more architectural in the sense that they describe a relationship between entities which could be seen as an infrastructure for a complete application, rather than a relationship which is based solely on inheritance. About the integration of frameworks, design patterns do not give us the context in which the framework is expected to be used. Finally about the evolution issue, it is possible to observe a change of design patterns without having a real evolution of the architecture. Thus design patterns are not a good gauge for following the evolution of a framework.

The main problems are incompleteness, because design patterns can not document all the design decisions emanating from the requirements, and its fine granularity, because design patterns overly describe the system, without revealing a more global view of the system.

[Ric98] believes that unlikely one notation or method can be used to express all aspects of an application's architecture. To cope with this, she proposes a variety of complementary forms of documentation to understand the what - the context: what is the application domain, what are its main elements and their interrelationships, what problems must be tackled in building the framework architecture - and the how - the solution: how the framework designers have chosen to tackle these problems.

- *Application domain model*: as a result of a domain analysis, it is necessary to identify domain concepts and making the relationships between them explicit.
- *Design space for the domain*: each problem in the domain can be solved using a variety of approaches. In a design space each problem represents an axis along which are possible or recommended solutions [DMNS97]. Understanding the design space allows us to situate the particular framework as a point in this space
- *Examples*: they illustrate what kinds of applications can be created using the framework. They show the variability in the framework.
- *Architectural patterns*: they present us with a metaphor - client/server, pipe/filter, etc - for architectural view, or a large part of one. Such metaphors can succinctly describe components, the relationships between them, and at the same time provide us with familiar scenarios for their behavior.
- *Scenarios*: They are instances of use-cases. They show how several components (i.e. granularity greater than classes) interact to assure one variation of the system's functionality.

These forms of description aim to give an understanding of the system at a coarse granularity and can complement more detailed forms of documentation such as design pattern and class hierarchies.

1.3 Architectural Description Language as an alternative formal approach

All the proposed approaches are focused on informal techniques. In spite that they can be efficient enough to communicate design decisions, they have limitations to represent the real semantics of different parts of a framework. [AAG93] explains that the imprecision produced by box-line drawings makes it difficult to attach unambiguous meanings to the descriptions. It may be difficult to know when an implementation agrees with the more abstract description. It is virtually impossible to reason formally about the descriptions. It is difficult to compare two different descriptions even for the same interpretation.

Thinking in terms of giving meaning to the descriptions of software systems, Architectural Description Languages (ADLs) have been proposed to support architecture-based development, formal modelling notations and analysis and development tools that operate on architectural specifications [MT97]. ADL must be able to communicate the architectural structures involved within a system to all stakeholders. The level of granularity, or abstraction, must be flexible enough to allow descriptions in sufficient detail or abstraction dependent on the users of the architectural description [All97].

The benefits of an architectural analysis are enhanced by precise semantics. Elimination of ambiguity is paramount in any architectural description to accurately describe a system. This requirement

must be balanced with the competing goals of allowing informal descriptions [Bro00].

This dissertation works with Wright ADL [All97] to infer a possible architectural description of some object oriented frameworks, whose design and use documentation is composed of a set of informal artifacts or the code itself. We do not claim that the described techniques in the previous section are incomplete. In fact our approach searches to be an union of complementary methodologies (Wright ADL [All97] and architectural patterns) and also represent a complement with the rest of techniques to give a semantic description, and avoiding as much as possible the ambiguity of box-line drawings in the documentation.

1.4 Related Work

The particular combination in the use of formal languages to describe an object oriented framework is only shown in [SG99]. In this work, they develop a specification of Sun's Enterprise Java-Beans. Firstly, they show formal architectural models based on protocols clarifying the intent of an integration framework, as well as exposing critical properties of it. Secondly, they describe techniques to create the model, and structure it to support traceability, tractability, and automated analysis. This work is a good approximation on ways to provide formal architectural models of object oriented frameworks.

Another related area is research on the analysis of architectural standards. In [AGI98] they looked at the high level architecture (HLA) for distributed simulation. HLA defines an integration standard for multi-vendor distributed simulations. They demonstrated that Wright could be used to model this framework and eliminate potential flaws in the HLA design.

Another area is work on protocol specification and analysis. There has been considerable research on ways to specify protocols using a variety of formalisms, including I/O Automata [LT89], SMV [CES85], SDL [Hol90] and Petri Nets [Pet77]. Most protocol analysis assumes one is starting with a complete description of the protocol. The problem is then to analyze that protocol for various properties. In contrast, in architectural modelling of systems, protocols are typically implicit in the APIs described in the framework documentation. Discovering what the protocols are, and how they determine the behaviour of the system is itself a key challenge.

Considering that the comparison between different architectural description languages can be made using benchmark problems, [AG96] proposes AEGIS Weapons System as a candidate problem. In this work, they show that while (architecturally speaking) the constructed system was relatively simple –less than a dozen architectural components- during the course of construction, it raised a surprisingly number of thorny architectural problems for the system implementors and integrators. Using Wright ADL [All97], they show that an architectural formalism helps to expose and solve some of the architectural problems.

With a different viewpoint and closer to the approach of this dissertation, [Men00] proposes a logic-meta programming to develop an expressive architectural model, algorithm and prototype tool for automatically checking conformance of the implementation of a software system to one of more architectural views. This approach is confined to static conformance checking only because the reasoning is about the static structure of a software implementation, and does not take run-time information into account.

1.5 Approaches and Contribution

This dissertation is concerned to get an architectural description about object oriented frameworks, whose documentation is based on informal techniques or the documentation is the code itself, using the Wright ADL [All97] as the main tool and architectural patterns as a complementary one. To fulfill this requirement, the following steps are made:

- ⇒ Firstly, all the features related to the concepts of software architectures, ADLs and formal approaches of description are studied. Thus, we get a complete context of the elements that we apply and use in the frameworks.
- ⇒ Secondly, main features of building/using an object oriented framework are also studied. We need to know how the frameworks are thought and how they evolve, in this way, we obtain the elements and the relationships that are represented (if possible) with the formal approaches of description. We also study documentation made with design and/or code itself of real frameworks.
- ⇒ The third step is the mapping from the information about the frameworks to an architectural description. If possible, we map also to an architectural pattern to see if this mapping can give us more information about the studied frameworks. Thus, we define a set of rules that an architect can follow to infer a possible architectural description. To do this last step the developed approach will consider some ideas presented in [LK94]. In this approach, they identify different levels of reuse and abstraction in a framework. Our approach considers different levels of abstraction to get an architectural description of the different parts of a framework and successively refinements from general to specific aspects to be very close to the implementation
- ⇒ The fourth step consists of analyzing the results from the third step to know what properties we could get using formal approaches of description applied on frameworks. We also consider if the formal approaches can be a complement to more known informal techniques of description
- ⇒ The fifth step is the study of evolution impacts on an architectural description of an object oriented framework considering the developed approach.

The contributions of this dissertation are:

- ⇒ The definition of rules to define a mapping from informal documentation of frameworks to an architectural description. Thus, firstly, the path from documentation and/or code to an architectural description of a framework is reduced to a recipe. Secondly the mapping allows us identify microarchitectures and architectural components and connectors related to the framework. Finally, we obtain a generic software architecture that captures the family of applications resulting from the framework instantiation.

- ⇒ The analysis of how efficient an ADL is to 'capture' the ambiguities caused by informal techniques.
- ⇒ The focus on a formal approach only can be a 'wrong' step in the idea of avoiding ambiguous descriptions
- ⇒ The proof that an ADL can be a complement to other informal and formal techniques. Thus, we get a uniform model of frameworks description that allows the users of the framework understand the abstract design of the underlying framework as well as the internal structure of its classes, in order to adapt and extend them.
- ⇒ The analysis of the evolution impacts of frameworks in the architectural descriptions.

1.6 Organization of the Thesis

This dissertation is organized in the following way: Chapter 2 describes the main features and formal approaches of description of software architectures. Chapter 3 describes general features about frameworks. Chapter 4 presents a previous analysis of problems found during the process of searching common points between software architectures and object oriented frameworks. Based on this analysis, Chapter 5 presents the developed mapping with the algorithms and two cases studies with their results. Chapter 6 shows a discussion of some important aspects of the developed approach and a discussion related to the results obtained in the previous chapter. Chapter 7 shows the evolution impacts of frameworks in resulting formal description. Finally, Chapter 8 presents the conclusions and future work.

Chapter 2: Software Architectures and Approaches of Description

During last ten years software architecture has begun to emerge as an important field of study for software engineering practitioners and researchers. This emergence is evidenced by a large body of work in areas such as domain specific architectures, architectural description languages, formal underpinnings for architectural design, and architectural design environments. To understand what aspects and what tools are being addressed, this chapter intends to give the reader a complete overview of this discipline.

We start clarifying the definition of software architectures to keep one of them that it is considered in the rest of this dissertation. Afterwards, we define and present the most known classical architectural patterns and styles, and we show their features and the context where they are applied. These conditions are important for this dissertation because we intend to identify them in the studied object oriented frameworks. Finally we show the main characteristics and properties of ADLs to describe software architectures and we also present Wright ADL (our work tool) analyzing which features of an 'ideal' ADL the language fulfills and which not.

2.1 Definitions of Software Architecture

The main problem in the field of software architecture is the lack of a clear definition of what an architecture is. Let's see the different approaches found in the literature. [GS92] considers that software architecture is emerging as a significant and different design level. According to [LC98] software architecture may then be basically defined as a description of the structural and dynamic properties of a system at a high level of abstraction.

But the software architecture is also concerned with issues such as the global control structure of a system, its main design elements and the way they share functionalities, the way design elements communicate (protocols for communication, synchronization, data access), the physical distribution of design elements, scaling and performance. Following this approach, one of the earliest definitions can be found in [PW92], where the software architecture is modeled with elements, form and rationale. Elements are either processing elements (e.g. procedures, filters), data elements (e.g. set of global variables), or connecting elements (e.g. procedure calls, messages). A form is defined by the constraints on the elements and the rationale captures the motivation for the choice of elements, form and architectural style. There is a more restrictive definition of software architecture given in [GS92], which only involves computational components and a description of interaction between these components which respectively correspond to processing elements and connecting elements and some part of form as defined in [GS92]. A more practical definition is the following : an abstract system specification consisting primarily of functional components described in terms of their behaviours and interfaces and interactions between components [HR94]. A fourth definition is extracted from [GP95] and synthesizes the previous ones : a software architecture is the structure of the components of a program/system, their interrelationships (often called connectors [SG96]), and principles and guidelines governing their design and evolution over time.

Clearly, from all the definitions given previously, a fact is software architecture focuses on raising the level of abstraction at which developers can reason about their systems. A system's architecture provides a model of the system that suppresses implementation detail and increases

the independence of system components, allowing many issues to be localized. Then the architecture can concentrate on the analyses and decisions that are most crucial to the system structure [All97].

An important remark is that a feature that distinguishes of software architecture from other views of the system is the explicit modelling of connectors. While components are the computational entities, connectors mediate and govern interactions among components. In this way connectors separate computation from communication, minimise component interdependencies and facilitate system understanding, analysis and evolution [MT97].

This dissertation considers the concepts of components and connectors explicitly to represent a framework architecture. The proposed approach works on different levels of abstraction refining the components and connectors, allowing refinement views of a possible software architecture and identifying architectural patterns or styles and proposing new ones (if necessary) which represent the family of applications that can be obtained instantiating the framework.

2.2 Architectural Patterns

In this section we introduce some architectural patterns and some general features. As we said previously, we propose to identify their existence in the frameworks. To do this, it is necessary to know the context where they can be applied. We will not discuss in detail about each of them. Detailed information can be found in [BMR+96].

Architectural patterns express fundamental structural organization schemas for software systems. They provide a set of predefined subsystems, specify their responsibilities, and include rules and guidelines for organizing the relationships between them.

Architectural patterns represent the highest-level patterns in the pattern system presented in [BMR+96]. They help you to specify the fundamental structure of an application. Every development activity that follows is governed by this structure -for example, the detailed design of subsystems, the communication and collaboration between different parts of the system, and its later extension. Each Architectural Pattern helps you to achieve a specific global system property, such as the adaptability of the user interface. Patterns that help to support similar properties can be grouped into categories. [BMR+96] proposes four categories:

- **From mud to structure:** patterns in this category help you to avoid a 'sea' of components or objects. In particular, they support a controlled decomposition of an overall system task into cooperating subtasks.
 - *Layers pattern* helps to structure applications that can be decomposed into groups of subtasks in which each group of subtasks is at a particular level of abstraction.
 - *Pipes and Filters pattern* provides a structure for systems that process a stream of data. Each processing step is encapsulated in a filter component. Data is passed through pipes between adjacent filters. Recombining filters allows you to build families of related systems.
 - *Blackboard pattern* is useful for problems for which no deterministic solution strategies are known. In Blackboard several specialized subsystems assemble their knowledge to build a possibly partial or approximate solution.
- **Distributed Systems:** The Broker pattern provides a complete infrastructure for distributed applications. The Microkernel and Pipes and Filters patterns only consider distribution as a secondary concern.
 - *Pipes and Filters pattern:* This pattern is more often used for structuring the functional core of an application than for distribution.
 - *Microkernel pattern* applies to software systems that must be able to adapt to changing system requirements. It separates a minimal functional core from

- extended functionality and customer-specific parts. The microkernel also serves as a socket for plugging in these extensions and coordinating their collaboration. Microkernel systems employ a Client-Server architecture in which clients and servers run on top of the microkernel component. The main benefit of such systems, however, is in design for adaptation and change.
- o *Broker pattern* can be used to structure distributed software systems with decoupled components that interact by remote service invocations. A broker component is responsible for coordinating communication, such as forwarding requests, as well as for transmitting results and exceptions.
 - **Interactive Systems:** Model-View-Controller and Presentation-Abstraction-Control patterns. Both patterns support the structuring of software systems that feature human-computer interaction.
 - o *Model-View-Controller pattern* divides an interactive application into three components. The model contains the core functionality and data. Views display information to the user. Controllers handle user input. Views and controllers together comprise the user interface. A change-propagation mechanism ensures consistency between the user interface and the model.
 - o *Presentation-Abstraction-Control pattern* defines a structure for interactive software systems in the form of a hierarchy of cooperating agents. Every agent is responsible for a specific aspect of the applications's functionality and consists of three components: presentation, abstraction, and control. This subdivision separates the human-computer interaction aspects of the agent from its functional core and its communication with other agents.
 - **Adaptable Systems:** Reflection and Microkernel patterns strongly support extension of applications and their adaptation to evolving technology and changing functional requirements.
 - o *Microkernel pattern* applies to software systems that must be able to adapt to changing system requirements. It separates a minimal functional core from extended functionality and customer-specific parts. The microkernel also serves as a socket for plugging in such extensions and coordinating their collaboration.
 - o *Reflection pattern* provides a mechanism for changing structure and behavior of software systems dynamically. It supports the modification of fundamental aspects, such as type structures and function call mechanisms. In this pattern, an application is split into two parts. A meta level provides information about selected system properties and makes the software self-aware. A base level includes the application logic. Its implementation builds on the meta level. Changes to information kept in the meta level affect subsequent base-level behavior.

In the approach presented in this dissertation, the architectural patterns will be considered in the top level of the description. As we said previously, we will work in different levels of abstraction and description, refining each level. The architectural patterns are useful in the topmost level because they give an global overview of an architecture that has to be specified and refined. [BMR+96] says that the selection of an architectural pattern should be driven by the general properties of an application at hand. But most software systems, however, cannot be structured according to a single architectural pattern. They must support several system requirements that can only be addressed by different architectural patterns. For example, you have to design both for flexibility of component distribution in a heterogeneous computer network and for adaptability of their user interfaces. You can combine Broker and Model-View-Controller patterns. The Broker pattern provides the infrastructure for the distribution of components, while the model of the MVC pattern plays the role of a server in the Broker infrastructure. Similarly, controllers take the roles of clients, and views combine the roles of clients and servers, as clients of the model and servers of the controllers. In the approach developed in this dissertation, we are inferring a framework architecture based on design and code, and in fact we map possible architectural patterns which

fits better to our proposed architecture. But not all the cases will allow us to map to them because the selection of an architectural pattern, or a combination of several, is only the first step when designing the architecture of a system. And our approach is more oriented to use design decisions that already have been taken, and if they did not assume any architectural pattern, it will be a difficult task to detect and infer them.

2.3 Architectural Styles

While a software architecture is defined by a layout of architectural elements, an architectural style consists of a set of shared assumptions and constraints across a family of software architectures [GCBA95]. The interest of a particular style is related to its ability to encapsulate important classes of design decisions and to emphasize important constraints on architectural elements.

Several architectural styles have already been identified. We cite the most known and their features. We will not give any details except their architectural elements and their invariants. All the details can be obtained from [SG96], [PW92]. We also omit advantages and disadvantages of each style, because they will be explained when applying in the inferred architectures during this dissertation. To have a more graphical idea of the styles, the Figure 2.1 shows some of them:

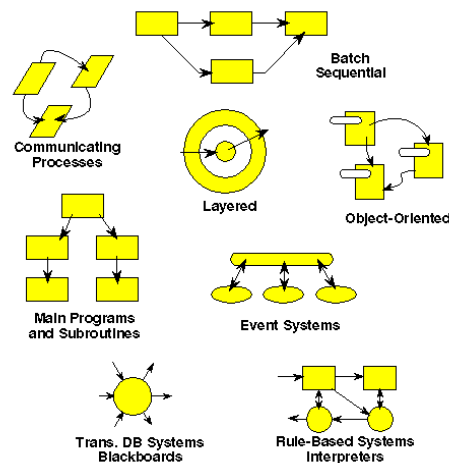


Figure 2.1: Architectural Styles in graphic notation

- **Pipes and Filters:** Each component (filters) has a set of inputs and a set of outputs. A component reads streams of data on its inputs and produces streams of data on its outputs, delivering a complete instance of the result in a standard order (through pipes). This is usually accomplished by applying a local transformation to the input streams and computing incrementally so output begins before input is consumed. The invariants are that the filters must be independent entities, they should not share state with other filters; and do not know the identity of their upstream and downstream filters.
- **Data Abstraction and Object-Oriented Organization:** data representations and their associated primitive operations are encapsulated in an abstract type or object. The components are the objects called as manager because they are responsible for preserving the integrity of a resource (representation). Objects interact through function and procedure invocations. Their invariants are that an object is responsible for preserving the integrity of its representation (usually by maintaining some invariant over it); and the representation is hidden from other objects.
- **Event-based, Implicit Invocation:** the components in this style are modules whose interfaces provide both a collection of procedures and a set of events. Procedures may

be called in the usual way. But in addition, a component can register some of its procedures with events of the system. This will cause these procedures to be invoked when those events are announced at run time. Thus the connectors in an implicit invocation system include traditional procedure call as well as bindings between event announcements and procedure calls. The invariant: The announcers of events do not know which components will be affected by those events. Thus components can not make assumptions about order of processing, or even about what processing, will occur as a result of their events.

- **Layered Systems:** A layered system is organized hierarchically, each layer providing service to the layer above it and serving as a client to the layer below. In some layered systems inner layers are hidden from all except the adjacent outer layer, except for certain functions carefully selected for export. The connectors are defined by the protocols that determine how the layers will interact. Topological constraints include limiting interactions to adjacent layers.
- **Repositories:** In this style there are two quite distinct kinds of components: a central data structure represents the current state, and a collection of independent components operate on the central data store. Interactions between the repository and its external components can vary significantly between systems. The choice of control discipline leads to major subcategories. If the types of transactions in an input stream of transactions trigger selection of processes to execute, the repository can be a traditional database. If the current state of the central data structure is the main trigger of selecting processes to execute, the repository can be a blackboard.
- **Table Driven Interpreters:** In an interpreter organization a virtual machine is produced in software. An interpreter includes the pseudo-program being interpreted and the interpretation engine itself. The pseudo-program includes the program itself and the interpreter's analog of its execution state (activation record). The interpretation engine includes both the definition of the interpreter and the current state of its execution. Thus an interpreter generally has four components: an interpretation engine to do the work, a memory that contains the pseudo-code to be interpreted, a representation of the control state of the interpretation engine, and a representation of the current state of the program being simulated.
- There are numerous other architectural styles. Some are widespread and others are specific to particular domains. Examples: Distributed Processes, Main Program/Subroutine Organizations, State Transition Systems, Process Control Systems, Multi-Phase Architectural Pattern (Figure 2).

Example: Multiple-Phase Architectural Style [PW92]

The multi-phase architectural style consists of processing elements and data elements that are exchanged between processing elements. If the multi-phase architectural style is organized sequentially, it also uses following connecting elements. This style can be used to describe a compiler. In this case,

Processing elements: lexer, parser, semantor, optimizer, code generator.

Data elements: characters, tokens, phrases, correlated phrases, annotated phrases, object code.

Connecting elements: procedure calls and parameters.

The restrictions are: the optimizer and the annotated phrases must be found together, but they are only preferred elements and not mandatory; and the lexer is constrained to accept a sequence of characters C, to produce a sequence of tokens T, and to preserve the ordering correspondence between characters and tokens:

Lexer: $C \rightarrow T$, where T preserves C.

Figure 2.2: Multi-Phase Architectural Style [PW92]

Observing the features of the different architectural styles, we can enumerate the basic properties they have:

- ⇒ They provide a vocabulary of design elements -component and connector types such as pipes, filters, clients, servers, parsers, databases, etc. [Gar95].
- ⇒ They define a set of configuration rules -or topological constraints -that determine the permitted compositions of those elements. For example, the rules might specify that a client-server organization must be an n-to-n relationship, or define a specific compositional pattern such as a pipelined decomposition of a compiler [Gar95].
- ⇒ They define a semantic interpretation, whereby compositions of design elements, suitably constrained by the configuration rules, have well-defined meanings [Gar95].
- ⇒ They define analyses that can be performed on systems built in that style. For example, a deadlock for client-server message passing [RJFC94].

Generally speaking, an architectural style expresses a particular kind of fundamental structure for a software system together with an associated method that specifies how to construct it. An architectural style also comprises information about when to use the architecture it describes, its invariants and specializations, as well as the consequences of its application.

We have cited 'pure' architectural styles. It is important to understand the individual nature of each of these styles, most systems typically involve some combination of several styles. There are different ways in which architectural styles can be combined [Gar95]:

- ⇒ *Through hierarchy*: A component of a system organized in one architectural style may have an internal structure that is developed a completely different style. For example, in an UNIX pipeline the individual components may be represented internally using another pipe and filter system.
- ⇒ *To allow a single component to use a mixture of architectural connectors*. For example, a component might access a repository through part of its interface, but interact through pipes with other components in a system, and accept control information through another part of its interface. In fact, UNIX pipe and filter systems do this, the file system playing the role of the repository and initialization switches playing the role of control.
- ⇒ To completely elaborate *one level of architectural description* in a completely different architectural style.

2.3.1 Practical Benefits of Architectural Styles

The principled use of architectural styles has a number of practical benefits [Gar95].

- ⇒ it promotes design reuse: routine solutions with well-understood properties can be reapplied to new problems with confidence.
- ⇒ It can lead to significant code reuse: often the invariant aspects of an architectural style lend themselves to shared implementations.
- ⇒ It is easier for others to understand a system's organization if conventionalized structures are used. For example, characterization of a system as a layered organization immediately conveys a strong image of the kinds of pieces which collaborates through services in different levels of the system.
- ⇒ Constraining the design space, an architectural style often allows specialized, style-specific analyses. For example, it is possible to analyze pipe-filter systems for schedulability, throughput, latency, and deadlock-freedom

2.4 Architectural Styles and Patterns

Architectural styles are very similar to the architectural patterns presented in [BMR+96]. In fact, every architectural style can be described as an architectural pattern. For example, the Multi-

phase architectural style [PW92] corresponds to the Pipes and Filters pattern [BMR+96]. On the other hand, architectural styles differ from patterns in several important respects:

Architectural styles only describe the overall structural framework for applications. Patterns for software architecture, however, exist in various ranges of scale, beginning with patterns for defining the basic structures of an application (architectural patterns) and ending with patterns that describe how to implement a particular design issue in a given programming language (idioms) [BMR+96].

Architectural styles are independent of each other, but a pattern depends on the smaller patterns it contains, on the patterns with which it interacts, and on the larger patterns in which it is contained [Ale76]

Patterns are more problem-oriented than architectural styles. Architectural styles expresses design techniques from a viewpoint that is independent of an actual design situation. A pattern expresses a very specific recurring design problem and presents a solution to it, all from the viewpoint of the context in which the problem arises [BMR+96].

2.5 Significance of Software Architecture to Software Engineering

A principled use of software architecture can have a positive impact on at least four aspects of software development:

- **Understanding:** Software Architecture simplifies our ability to comprehend large systems by presenting them at a level of abstraction in which all the system can be understood [GS92] [PW92]. Moreover, architectural description exposes the high level constraints on system design as well as the rationale for making specific architectural choices [GP94].
- **Reuse:** Architectural description support reuse at multiple levels. While most current work on reuse focuses on component libraries, architectural design supports, in addition, both reuse of large components (such as subsystems), and also the complementary need for reusable frameworks into which components can be integrated [GP94].
- **Evolution:** Software architecture can expose the dimensions along which a system is expected to evolve. By making explicit what are the "load-bearing walls" of a system, system maintainers can better understand the ramifications of changes, and thereby more accurately estimate costs of modifications [PW92].
- **Analysis:** Architectural description provides new opportunities for analysis [PW92], including high-level forms of system consistency checking [AG94], conformance to an architectural style [AAG93], and domain-specific analysis for architectures that conform to specific styles [GP94].

2.6 Architecture Description Languages (ADLs): General features

In this dissertation, we use Wright ADL [All97] to propose an architectural description of a framework. Firstly, we discuss general features about ADL, and after we describe briefly the elements and ways of working with Wright ADL [All97].

This field seems to be another item in the list of unclear definitions. Nobody agrees in the research community on what is an ADL, what aspects of an architecture should be modeled in an ADL, and which ADL fits for a particular problem.

ADLs have been proposed as the answer to support architecture-based development, formal modelling notations and analysis and development tools that operate on architectural specifications.

An ADL is a language that provides features for modeling a software system's conceptual architecture. A number of ADLs have been proposed for modeling architectures both within a particular domain and as general-purpose architecture modeling languages, for example, Aesop [GAO94], MetaH [Ves96], LILEANNA [Tra93], ArTek [TLPD95], C2 [MORT96, MTW96], Rapide [LKA+96, LV95], Wright [All97], UniCon [SDK+95], Darwin [MDEK95, MK96], and SADL [MQR95].

ADLs provide a concrete syntax and a conceptual framework for characterizing architectures [GMW97]. The building blocks of an architectural description are : components, connectors, and architectural configurations (or topologies). An ADL must provide the means for their explicit specification. An ADL typically subsume a formal semantic theory. That theory is part of an ADL's underlying framework for characterizing architectures ; it influences the ADL's suitability for modeling particular kinds of systems (e.g. highly concurrent systems) or particular aspects of a given system (e.g., static properties).

2.6.1 Architectural Elements and Its Properties

ADLs must provide components, connectors and architectural configurations. [MT97] have developed a framework for classifying and comparing ADLs. In this work, they enumerate several aspects of both components and connectors that are desirable, but not essential : interfaces (for the connectors) and types, semantics, constraints, and evolution (both). Desirable features of configurations are understandability, heterogeneity, compositionality, constraints, refinement and traceability, scalability, evolution and dynamism.

Let's see all the elements and properties in general terms and after we determine which properties are fulfilled by Wright ADL [All97].

1. Components : unit of computation or data store. Therefore, components are loci of computation and state [SDK+95]. A component in an architecture may be as small as a single procedure (e.g. MetaH procedures) or as large as an entire application (e.g. hierarchical components in C2 and Rapide or macros in MetaH). It may require its own data and/or execution space, or it may share them with other components.

- Interface : a component's interface is a set of interaction points between it and the external world. An interface specifies the services (messages, operations, and variables) a component provides.
- Types: ADLs can support reuse by modeling abstract components as types and instantiating them multiple times in an architectural specification.
- Semantics: Modelling of component semantics enables analysis, constraint enforcement, and mapping of architectures across levels of abstraction.
- Constraints: Properties or assertions must be specified to ensure adherence to intended component uses, enforce usage boundaries, and establish intra-component dependencies.
- Evolution: As components evolve, ADLs should support component evolution through subtyping and refinement.

2. Connectors are architectural building blocks used to model interactions among components and rules that govern those interactions. Unlike components might not correspond to compilation units in implemented systems.

- Interfaces : In order to enable proper connectivity of components and their communication in an architecture, a connector should export as its interface those services it expects. Therefore, a connector's interface is a set of interaction points between it and the components attached to it. It enables reasoning about the well-formedness of a configuration.

- **Types:** As architecture-level communication is often expressed with complex protocols, ADLs should model connectors as types to abstract away these protocols and make them reusable.
- **Semantics:** Architectural descriptions should provide connector protocol and transaction semantics to perform analyses of component interactions, consistent refinements across levels of abstraction, and enforcement of interconnection and communication constraints.
- **Constraints:** Connector constraints must be specified to ensure adherence to interaction protocols, establish intra-connector dependencies, and enforce usage boundaries.
- **Evolution:** Component interactions are governed by complex and changing protocol. Maximizing connector reuse is achieved by modifying or refining existing connectors, ADLs can support connector evolution with subtyping and refinement.

3. **Configurations** : Architectural configurations, or topologies, are connected graphs of components and connectors that describe architectural structure. This information is needed to determine whether : appropriate components are connected, their interfaces match, connectors enable proper communication, and their combined semantics result in desired behavior. In concert with models of components and connectors, descriptions of configurations enable assessment of concurrent and distributed aspects of an architecture, e.g. potential for deadlocks and starvation, performance, etc. Configurations also enable analyses for adherence to design heuristics and style constraints.

- **Understandable Specifications:** ADLs must model structural information with simple and understandable syntax, where system structure is clear from a configuration specification alone.
- **(Hierarchically) Compositionality:** Architectures may be required to describe software systems at different levels of detail, where complex behaviours are either explicitly represented or abstracted away into single components and connectors. An ADL may also need to support situations in which an entire architecture becomes a single component in another, larger architecture.
- **Heterogeneity:** A goal of architectures is to facilitate development of large systems, with components and connectors of varying granularity, implemented by different developers, and in different programming languages. ADLs should provide facilities for architectural specification and development with heterogeneous components and connectors.
- **Constraints:** Constraints that depict dependencies among components and connectors in a configuration are as important as those specific to individual components and connectors. Many global constraints are derived from or directly dependent upon local constraints.
- **Refinement and Traceability:** ADL must enable correct and consistent refinement of architectures to executable systems and traceability of changes across levels of refinement.
- **Scalability:** Architectures are intended to support large-scale systems. ADLs must support specification and development of systems that may grow in size.
- **Evolution:** Architectures evolve to reflect evolution of a single software system; they also evolve into families of related systems. ADLs need to augment evolution support at the level of components and connectors with features for incremental development and support for system families. Incrementality of an architectural configuration can be viewed from two different perspectives. One is its ability to accommodate addition of new components, and the another one is the ADL's support for incomplete architectural descriptions. Another aspect of evolution is support for application families.
- **Dynamism:** Configurations exhibit dynamism by allowing replication, insertion, removal, and reconnection of architectural elements or subarchitectures during execution.

2.6.2 Wright: An Architectural Description Language

2.6.2.1 Structure

Wright is a formal language for describing software architecture. As with most ADLs, Wright describes the architecture of a system as a graph of components and connectors. Components represent the main centers of computation, while connectors represent the interactions between components. While all architecture description languages allow the specification of new component types, unlike many languages, Wright also supports the explicit specification of new architectural connector types [All97].

To illustrate, let's have a look at a configuration to describe a client-server system:

```

Configuration Simple Example
  Component Server =
    Port provide = [provide protocol]
    Computation = [Server computation]
  Component Client =
    Port request = [request protocol]
    Computation = [Client protocol]
  Connector C-S-Connector =
    Role Client = [Client protocol]
    Role Server = [Server Protocol]
    Glue = [Glue Protocol]

Instances
  s : Server
  c : Client
  cs : C-S-Connector

Attachments
  s.provide as cs.server
  c.request as cs.client
end SimpleExample

```

. This example shows three basic elements of a Wright system description : component (*Server and client*) and connector (*C-S-Connector*) type declaration , instance declarations, and attachments. The instance declarations and attachments together define a particular system configuration. Let's see briefly some details about each element in the structure of a configuration

1. In Wright the description of a component has two important parts, the interface and the computation. A component interface consists of a number of ports. Each port defines a point of interaction through which the component may interact with its environment.
2. A connector represents an interaction among a collection of components. For example, a pipe represents a sequential flow of data between two filters. A Wright description of a connector consists of a set of roles and the glue. Each role defines the allowable behavior of one participant in the interaction. A pipe has two roles, the source of data and the recipient. The glue defines how the roles will interact with each other.
3. The description of the components and connectors are considered as types that represent the properties of these artefacts. Thus "Client" is a type of component, while there may be many instances of a client in a given system. Wright requires that each instance be explicitly and uniquely named and provides the *instances* declarations to distinguish the different instances of each component and connector type that appear in the configuration.
4. Once the instances have been declared, a configuration is completed by describing the *attachments* . The attachments define the topology of the configuration, by showing which components participate in which interactions. This is done by associating a component's port with a connector's role. For example, the attachment declaration *s.provide as cs.server*

indicates that the component *s* will play the role of server in the interaction *cs*. It will find this role through the port *provide*.

- The attachments declarations bring together each of the elements of an architectural description. The component carries out a *Computation*, part of which is a particular interaction, specified by a *Port*. That port is attached to a *Role*, which indicates what rules the port must follow in order to be a legal participant in the interaction specified by the connector. If each of the components, as represented by their respective ports, obeys the rules imposed by the roles, then the connector *Glue* defines how the *Computations* are combined to form a single, larger computation.

Wright supports also hierarchical descriptions. In particular, the computation of a component (or the glue of a connector) can be represented either directly by a behaviour specification or by an architectural description itself. In the latter case, the component serves as abstraction boundary for a nested architectural subsystem, which is described as a configuration in the same way as it was made previously. However, for a component the nested architectural description has an associated set of bindings, which define how the unattached port names on the inside are associated with the port name of the enclosing component. The figure 2.3 illustrates the use of hierarchy using the previous example and the hierarchical specification in Wright looks like that:

```

Configuration HierServer
  Connector CSConn
  Component ClientType
  Component ServerType
  Port Service ...
  Computation
    Configuration SecureData
      Component Coordinator
      Instances
        C: Coordinator
        Security : SecurityManager
      Attachments
        C.Secure as S1.Client
        Security.Service as S1.Service
      End SecureData
      Bindings
        C.Combined = Service
      End Bindings
    End Computation
  End HierServer
  
```

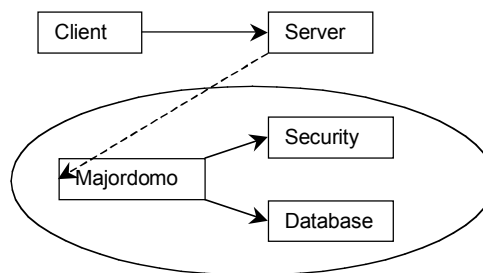


Figure 2.3: Hierarchical Structure

2.6.2.2 Style

Wright can be used to define architectural styles. A style defines a set properties that are shared by the configurations that are members of the style. These properties can include a common vocabulary and restrictions on the ways this vocabulary can be used in configurations.

In Wright, common vocabulary is introduced by declaring a set of component and connector types, using the declaration constructs introduced previously for instance descriptions.

For example, a client-server style would include a declaration of connector type Client. Then when a configuration is declared in the client-server style, Clients are automatically available for use.

Interfaces Types: in addition to declaring complete component and connector types, a style's properties may constrain only part of a component or a connector. For example, in the pipe-filter style all components are filters, which use only dataflow for input and output. This commonality of filters needs to be stated, while the computation itself will differ between different filters. An interface type declarations looks like the following one:

```
Interface Type DataInput = [read data repeatedly, closing the port at or before end-of-data]
Interface Type DataOutput = [write data repeatedly, closing the port to signal end-of-data].
```

Interfaces types can then be used either as the port of a component or as the role of a connector. In the latter case, the interface represents a constraint on the port interfaces that may be used in the role.

Parameterization: This ADL allows any part of the description of a type to be replaced with a *placeholder*, which is then filled with a parameter when the type is instantiated. So the type of a port or role, a computation, the name of an interface are all parameterizable. For example, in the Unix pipe-filter style, all components have one input, named Stdin, and two outputs, named Stdout and Stderr. The interface to all Unix filters is the same. But the computation performed by each Unix filter is different. We can describe the Unix filter as a parameterized component type, as follows:

```
Component Unix-filter (C: Computation)
  Port Stdin = DataInput
  Port Stdout = DataOutput
  Port Stderr = DataOutput
  Computation = C
```

We can then use this description to describe any number of Unix filters :

```
Upper : Unix-filter ([pass output, translating to uppercase])
Lower: Unix-filter ([pass output, translating to lowercase])
LaTex: Unix-filter ([translate input in .tex form to .dvi form; error messages are sent to Stderr])
```

Another way of parameterizing a type description is by number. Suppose, for example, that a particular class of filter system uses many filters that split their input among a number of outputs. For example,

```
Component SplitFilter (nout: 1..)
  Port Input = DataInput
  Port Output1..nout = DataOutput
  Computation = [read from Input repeatedly, writing to Output1, Output2, etc. in succession]
```

The number parameter can be used to control the number of particular kinds of ports or roles that can appear. A port or role description that can have multiple copies is indicated by

specifying a range of integers as a subscript to its name. In this example, there can be more than one *Output* port, depending on the value of the *nout* parameter.

Constraints: A Wright style description may declare properties that must be obeyed by any configuration in the style. For example, the Wright definition of the pipe-filter style would indicate that all the connectors must be pipes as follows: $\forall c: \text{Connectors} \bullet \text{Type}(c) = \text{Pipe}$.

In addition, the style would require that all components in the system use only *DataInput* and *DataOutput* ports:

$$\forall c: \text{Components}; p: \text{Port} \mid p _ \text{Ports}(c) \bullet \text{Type}(c) = \text{DataInput} \vee \text{Type}(c) = \text{DataOutput}$$

Each of the constraints declared by a style represents a predicate that must be satisfied by any configuration declared to be a member of the style. The notation for constraints is based on first order predicate logic. The constraints refer to the following sets and operators:

- *Components*: the set of components in the configuration.
- *Connectors*: the set of connectors in the configuration.
- *Attachments*: the set of attachments in the configuration. Each attachment is represented as a pair of pairs ((*comp*, *port*), (*conn*, *role*)).
- *Name(e)* : the name of event *e*, where *e* is a component, connector, port, or role.
- *Type(e)*: the type of element *e*.
- *Ports(e)*: the set of ports on component *e*.
- *Computation(c)*: the computation of component *c*.
- *Roles(c)*: the set of roles of connector *c*.
- *Glue(c)*: the glue of connector *c*

In addition, any type has been declared as part of the style's vocabulary may be referred to by name. As we saw in the example above, the pipe-filter style introduces *Pipe*, *Data Input*, and *DataOutput*, and the constraints of the style refer to these types by name. Let's see then the complete example.

Style Pipe-Filter

Connector *Pipe*

Role *Source* [*deliver data repeatedly, signalling termination by close*]

Role *Sink* [*read data repeatedly, closing at or before end of data*]

Glue [*Sink will receive data in same order delivered by Source*]

Interface Type *DataInput* = [*read data repeatedly, closing the port at or before end-of-data*]

Interface Type *DataOutput* = [*write data repeatedly, closing the port to signal end-of-data*].

Constraints

$$\forall c: \text{Connectors} \bullet \text{Type}(c) = \text{Pipe} \wedge$$

$$\forall c: \text{Components}; p: \text{Port} \mid p _ \text{Ports}(c) \bullet \text{Type}(c) = \text{DataInput} \vee \text{Type}(c) = \text{DataOutput}$$

2.6.2.3 Behaviour: Definition of interaction protocols

Each part of a Wright description –port, role, computation and glue– is defined using a variant of CSP [Hoa85]. Each such specification defines a pattern of events (called a process) using operators for sequencing (“_” and “;”), choice (“_” and “|”), and parallel composition (“||”).

Let's see briefly the main details about CSP

- *Processes and Events* : A process describes an entity that can engage in communication events. Events may be primitive or they can have associated data as *e?x* and *e!x*, representing input and output of data, respectively.
- *Prefixing* : A process that engages in event *e* and then becomes process *P* is denoted *e_P*
- *Sequencing* : (« sequential composition ») A process that behaves like *P* until *P* terminates (§) and then behaves like *Q*, is denoted *P;Q*

- *Interrupting* : A process that behaves like P until the occurrence of the first event in Q, is denoted P_Q .
- *Alternative* : (« external choice ») A process that can behave like P and Q, where the choice is made by the environment, is denoted $P \ _ \ Q$. (« Environment» refers to the other processes that interact with the process.)
- *Decision* : (« internal choice ») A process that can behave like P or Q, where the choice is made (non-deterministically) by the process itself, is denoted P_Q .
- *Named Processes* : Process names can be associated with a (possibly recursive) process expression. Processes may also be subscripted to represent internal state.
- *Parallel Composition* : Processes can be composed using the || operator. Parallel processes may interact by jointly (synchronously) engaging in events that lie within the intersection of their alphabets. Conversely, if an event e is in the alphabet of processes P1 and P2, then P1 can only engage in the event if P2 can also do so. That is, the process $P1 \ || \ P2$ is one whose behavior is permitted by both P1 and P2.

Wright extends CSP in three minor syntactic ways:

- it distinguishes between *initiating* an event and *observing* an event.
- it uses the symbol § to denote the successfully-terminating process. (In CSP this is usually written « SKIP »).
- It uses a quantification operator: $\langle op \rangle x:S \cdot P(x)$. This operator constructs a new process based on the process expression $P(s)$, and the set S, combining its parts by the operator $\langle op \rangle$. For example, $_ i : \{1,2,3\} \cdot Pi = P1_P2_P3$: i.e., a choice among one of three processes, P1, P2, P3. Similarly, $_ x:S \cdot P(x)$ is a process that consists of some unspecified sequencing of the processes:

$$_ x:S \cdot P(x) = _ x:S \cdot (P(x); (_ x:S \setminus \{x\} \cdot P(y))).$$

Thus, we can define the protocols for our example. A simple client role might be defined by the CSP process:

$$\mathbf{Role \ Client} = (\overline{request} \ _ \ result?x \ _ \ Client) \ _ \ \S$$

This specification defines a participant in an interaction that repeatedly makes a request and receives a result, or chooses to terminate successfully.

It is also possible to define interface types and to use them in the interaction protocol of the role, glue, port and computation. For example,

$$\mathbf{Interface \ Type \ ClientPullT} = \overline{open} \ _ \ \mathbf{Operate} \ _ \ \S$$

$$\text{where } \mathbf{Operate} = \overline{request} \ _ \ result?x \ _ \ \mathbf{Operate} \ _ \ \overline{close} \ _ \ \S$$

This specification defines the behaviour of a participant in an interaction that repeatedly opens a connection and makes a request and wait for the answer or chooses to terminate successfully.

$$\mathbf{Interface \ Type \ ServerPushT} = \overline{open} \ _ \ \mathbf{Operate} \ _ \ \S$$

$$\text{where } \mathbf{Operate} = \overline{request} \ _ \ result!x \ _ \ \mathbf{Operate} \ _ \ \overline{close} \ _ \ \S$$

This specification defines the behaviour of a participant in an interaction that waits for the opening of a connection and a request and sends a result or wait for a closing event to terminate successfully

Thus, the specification of the connector is given by the following structure:

$$\mathbf{Connector \ C-S-Connector}$$

$$\begin{aligned}
 &\mathbf{Role\ Client} = \mathbf{ClientPullT} \\
 &\mathbf{Role\ Server} = \mathbf{ServerPushT} \\
 &\mathbf{Glue} = \mathbf{Client.open} _ \overline{\mathbf{Server.open}} _ \mathbf{Glue} \\
 &_ \mathbf{Client.close} _ \overline{\mathbf{Server.close}} _ \mathbf{Glue} \\
 &_ \mathbf{Client.request} _ \overline{\mathbf{Server.request}} _ \mathbf{Glue} \\
 &_ \mathbf{Server.result\ ?x} _ \overline{\mathbf{Client.result!\ x}} _ \mathbf{Glue}
 \end{aligned}$$

The connector waits for the occurrence of the events in the client-side (in the first three cases) and notifies the server-side of this occurrence. In the last case, when the server sends a value, the client is notified of this event.

This small example intends to be illustrative to understand the notation which will be used in the rest of this dissertation. A more detailed of the use of CSP notation to define interaction protocols can be consulted in [All97].

2.6.2.4 Validation of Descriptions

As a formal specification language, Wright has value beyond enabling architects to write down an architectural description. Another important aspect of the language is its support for analysis and reasoning about the described system.

Two criteria for an architectural description are consistency and completeness. Informally, consistency means that the description makes sense; that different parts of the description do not contradict each other. Completeness is the property that a description contains enough information to perform an analysis; that the description does not omit details necessary to show a certain fact or to make a guarantee. Thus, completeness is *with respect to* a particular analysis or property.

Because the architectural level of design is fundamentally concerned with questions of structuring and composition, consistency among parts is especially critical at this level of design. The principles that ensure that a system will function as a coherent entity must be built in as part of the overall structure; if the abstract description is inconsistent, any refinement or implementation of it must necessarily retain that inconsistency.

The problem of completeness is especially critical for the architect because of the importance of abstraction at this level of design. There is always a tension between the need to include critical information that is necessary to guarantee important system properties and the risk of cluttering the architecture with constraints and details that can make the architecture unwieldy and difficult to work with.

To guarantee that an architectural description is both consistent and complete, Wright provides a set of tests. We just mention them. More detailed information can be obtained in [All97]. In our approach we use this set of tests to verify that the description that we get in our algorithm is valid exploiting the formal side of Wright ADL

1. Port/Computation Consistency (component)
2. Connector Deadlock-Free (connector)
3. Roles Deadlock-Free (role)
4. Single Initiator (connector)
5. Initiator Commits (any process)
6. Parameter Substitution (instance)
7. Range Check (instance)
8. Port-Role Compatibility (attachment)
9. Style Constraints (configuration)
10. Style Consistency (style)
11. Attachment Completeness (configuration)

2.6.2.5 Properties

Having a general overview of how to use Wright ADL, let's see which are the properties that this language according to [MT97]. Thus, this language:

- Formalizes the semantics of architectural connections.
- Models components and connectors at a high level of abstraction and do not assume or prescribe a particular relationship between an architectural description and an implementation. It is referred as *implementation independent*.
- Provides the specification of component interfaces through the concept of port
- Models component semantics allowing the specification of component functionality in CSP.
- Express the constraints of the components specifying the protocols of interaction for each port.
- Refers to connector interface points as roles.
- Models connectors as first-class entities.
- Models connector semantics allowing the specification of connector glue with CSP.
- Constrains connectors by specifying protocols for each role.
- Does not facilitate connector subtyping, but supports type conformance, where a role and its attached port may have behaviorally related, but not necessarily identical, protocols.
- Has the best potential to facilitate understandability of architectural structure.
- Does not provide a 'semantically-sound' graphical notations.
- Allows hierarchical composition in principle, but provide no specific constructs to support it.
- Has been highlighted as an example of ADL lacking scalability features, yet they have both been used to specify architectures of large, real world systems.

2.7 Summary

This chapter explains all the features related to software architectures and ADLs focusing on Wright ADL. Thus, we have the vocabulary of the context and tools that we will be used to infer an architectural description of an object oriented frameworks.

Chapter 3: Object Oriented Frameworks

Designing software is hard and design of reusable software is even harder. The object oriented software technology is not the exception in the problem. However the development of object oriented framework seems to cope with this trouble in some aspects. In this chapter, we introduced the concept of object oriented frameworks and their main characteristics. We focus our work on two kinds: application and component frameworks.

3.1 Definition

The literature about this field shows a set of possible definitions to characterize an object oriented framework. However we consider one given in [FSJ00] that defines a framework as a reusable design of a system that describes how the system is decomposed into a set of interacting objects. Sometimes the system is an entire application; sometimes it is just a subsystem. The framework describes both the component objects and how these objects interact. It describes the interface of each object and the flow control between them. It describes how the system's responsibilities are mapped onto its objects [Joh88, WBJ90]. The most important part of a framework is the way that a system is divided into its components [Deu89]. Frameworks also reuse implementation, but that is less important than reuse of the internal interfaces of a system and the way that its functions are divided among its components. This high-level design is the main intellectual content of software, and the frameworks are a way to reuse it.

3.2 Differences with other concepts

The possibility of the frameworks of being used as the base for the development of a number of applications in the application domain constrasts to the normal way of developing an object-oriented application. Considering this issue, let's enumerate briefly which are the differences of a framework compared to other concepts such as an object oriented design pattern, a pattern language and an ordinary object-oriented application. We only keep the concepts that appear during the development of this dissertation

An object oriented design pattern differs from a framework in three ways [GHJV95]. Firstly, the design patterns are more abstract than a framework since frameworks are embodied in code meanwhile only the examples of design patterns are coded. The design patterns also describe the intent, trade-offs, and consequences of a design, which is not the case for frameworks. Secondly, design patterns are smaller architectures than frameworks. Thus, a framework can contain a number of design patterns, but the opposite is never possible. Then the design patterns have no major impact of the application's architecture. Finally, frameworks are more specialized than design patterns. Frameworks are always related to a specific application domain, whereas design patterns are general and can be applied in any application domain.

Pattern languages differ from frameworks in the way that a pattern language describes how to make a design where an object oriented framework is a design. Pattern languages complement a framework since they can teach software engineers how to use a framework, and describe why it was designed the way it was. [Mat96]

An object oriented application differs from a framework in that the application describes a complete executable program that satisfies a requirement specification. The framework captures

the functionality of the application but it is not executable because it does not cover the behaviour in the specific application case.[Mat96]

3.3 Characterization of Frameworks by different dimensions

The most important dimensions to characterize are: the problem domain the framework addresses, the internal structure of the framework and how the framework is intended to be used [Tal94].

Considering the framework domains, they are classified as:

- **Application frameworks** are frameworks that cover functionality that can be applied to different domains. Examples of application frameworks are frameworks for graphical user interfaces [WG94].
- **Domain frameworks** capture knowledge and expertise in a particular problem domain. Frameworks for manufacturing control [Sch95] and multimedia are examples of domain frameworks.
- **Support frameworks** are frameworks that offer low-level system services such as device drivers [And94] and file access.

Support frameworks are typical cases of fine-grained frameworks, whereas application and domain frameworks often are monolithic frameworks.

Taking into account the framework structures, if the framework's internal structure is described it can make it easier to understand the behaviour of the framework. The internal structure of a framework is related to the concepts of software architectures. In [GS92] a number of common software architectures have been identified. [BMR+96] calls these 'architectural frameworks' since they have been designed in a way that captures the main structure of an object-oriented software architecture: The *Layered architectural framework*, the *Pipes Filters architectural framework*, the *Model View Controller architectural framework*, the *Presentation-Abstraction-Controller architectural framework*, the *Reflective architectural framework*, the *Microkernel architectural framework*, the *Blackboard architectural framework* and the *Broker architectural framework*. We just mentioned them, but in fact we consider them as 'architectural patterns' in fine-grained level compared to framework, that can be used integrated in an software architecture.

About the way of use issue, there are two possibilities [Ada95] [FSJ00]:

- ⇒ Architecture driven or inheritance focused frameworks (whitebox frameworks)
- ⇒ Data driven or composition focused frameworks (black box frameworks)

In the case of architecture driven or inheritance focused frameworks, the main approach is to develop applications relying on inheritance and dynamic binding. Framework users make their adaptation of the framework through:

1. inheriting from framework base classes and
2. overriding pre-defined hook methods using patterns like Template Method [GHJV95].

A problem with frameworks that are architecture-driven is that it can be difficult to adapt them because:

1. it requires application users to have an intimate knowledge of the frameworks' internal structure,
2. the framework users have to provide the implementation of the behaviour themselves, which implies a large amount of code to be written.

Thus, they tend to produce systems that are tightly coupled to the specific details of the framework's inheritance hierarchies.

However in data driven or composition focused frameworks, adapting the frameworks to the specific needs of the application means relying on object composition and delegation. Existing functionality is reused by

1. defining components that conform to a particular interface, and

2. integrating these components into the framework using patterns like Strategy [GHVJ95] and Functor.

How objects can be combined are described by the framework, but what the framework does depends on what objects the framework user passes into the framework. A framework that is data-driven is normally easy to use but limiting. They are more difficult to develop since they require frameworks developers to define interfaces and hooks that anticipate a wider range of potential use-cases [HJE95]

To handle the problems associated with using inheritance-focused and composition-focused frameworks, a suitable approach can be to provide the framework with an architecture-driven base and a data-driven layer so it is both extendable and easy to use [Tal94]. How the objects can be combined are described by the framework, but what the framework does depend on what objects the framework user passes into the framework. To handle the problems associated with using inheritance focused and composition focused frameworks, a suitable approach can be to provide the framework with an architecture-driven base and a data driven layer so it is both extendable and easy to use.

3.4 Application Frameworks: Specific Features

As our approach is devoted to be applied in application frameworks because we explore internal structures, let's then the main features presented in [FSJ00] that we intend to represent in an architectural description.

Each object in the framework is described by an abstract class. An abstract class is a class with no instances, so it is used only as a superclass [WBJ90]. An abstract class usually has at least one unimplemented operation deferred to its subclasses. Since an abstract class has no instances, it is used as a template for creating subclasses rather than as a template for creating objects. Frameworks use them as design of their components because they both define the interface of the components and provide a skeleton that can be extended to implement the components.

In addition to providing an interface, an abstract class provides part of the implementation of its subclasses. For example, a template method defines the skeleton of an algorithm in an abstract class, deferring some of the steps to subclasses [GHJV95]. Each step is defined as a separate method that can be redefined by a subclass, so a subclass can redefine individual steps of the algorithm without changing its structure. The abstract class can either leave the individual steps unimplemented (in other words, they are abstract methods) or provide a default implementation (in other words, they are hook methods) [Pre95]. A concrete class must implement all the abstract methods of its abstract superclass and may implement any of the hook methods. It will then be able to use all the methods it inherits from its abstract superclass.

Frameworks take advantage of all three of the distinguishing characteristics of object oriented programming languages: data abstraction, polymorphism, and inheritance. Like an abstract data type, an abstract class represents an interface behind which implementation can change. Polymorphism is the ability for a single variable or procedure parameter to take on values of several types. Object-oriented polymorphism lets a developer mix and match components, lets an object change its collaborators at runtime, and makes it possible to build generic objects that can work with a range of components. Inheritance makes it easy to make a new component. a framework describes the architecture of an object-oriented system; the kinds of objects in it, and how they interact. It describes how a particular kind of program, such as a user interface or network communication software is decomposed into objects. It is represented by a set of classes (usually abstract), one for each kind of object, but the interaction patterns between objects are just as much a part of framework as the classes.

The first widely used framework, developed in the late 1970s, was the Smalltalk-80 user interface framework called Model/View/Control (MVC) [GoI84, KP88, LP91]. MVC showed that object-oriented programming was well suited for implementing graphical user interfaces (GUIs). It divides a user interface into three kinds of components: models, views, and controllers. These

objects work in trios consisting of a view and a controller interacting with a model. A model is an application object and is supposed to be independent of the user interface. A view manages a region of the display and keeps it consistent with the state of the model. A controller converts user events (mouse movements and key presses) into operations on its model and view. For example, controllers implement scrolling and menus. Views can be nested to form complex user interfaces. Nested views are called subviews.

The important classes in the framework, such as Model, View, and Controller of MVC, are usually abstract. Like MVC, a framework usually comes with a component library that contains concrete subclasses of the classes in the framework. Although a good component library is crucial companion to a framework, the essence of a framework is not the component library, but the model of interaction and control flow among its objects.

3.5 Components and Frameworks

Considering white-box frameworks, we said that adapting them can be very difficult because one has to understand what each element in the structure means and how the elements work together in order to reuse the structure. The complexity of adaptation of course depends on the complexity of structure to be adapted. Moreover, putting several complex structures together to form a bigger system (e.g. merging together groups of statements or different class hierarchies) is also difficult. Thus, [MN95] proposes the use of the software component to solve this problem.

A component is an abstraction of a software structure that may be used to build bigger systems, while hiding the implementation details of the smaller structures. Putting together components is simple, since each component has a limited set of 'plugs' with fixed rules specifying how it may be linked with other components. Instead of having to adapt the structures of a piece of software to modify its functionality, a user plugs the desired behaviour into the parameters of the component.

There are two important aspects to components:

- Encapsulation of software structure as abstract components.
- Composition of components by binding their parameters to specific values, or other components.

Encapsulation is the means to achieve variability, since the possible variation is expressed in the parameters (or 'plugs') of the component. Adaptability is achieved during composition, since a software structure composed from components can be more easily reconfigured than an unencapsulated structure.

The open variability of white-box structures can be changed for the fixed variability of possible connections to the plugs of the component. This restriction of variability is possible due to the fixed intended purpose of the component; it also includes the possibility to check the correctness of combinations of parameters. This is called 'closed' or 'black-box' software reuse.

What are the differences between objects and components? First of all, objects encapsulate services, whereas components are abstractions that can be used to construct object-oriented systems. Objects have identity, state and behaviour, and are always run-time entities. Components, on the other hand, are generally static entities that are needed at system build-time. They do not necessarily exist at run-time. Components may be of finer or coarser granularity than objects: e.g. classes, templates, mix-ins, modules. Components should have an explicit composition interface, which is type-checkable. An object can be seen as a special kind of stateful component that is available at run-time.

A piece of software can be called a component if it has been designed to be composed with other components. In general this is done to address a particular class of applications. If that is the case, we say a component framework has been developed for that application area.

A component framework does not just consist of a library of components, but must also define a generic architecture for a class of applications. A component framework is a collection of software artefacts that encapsulated domain knowledge, requirements models, a generic software architecture and a collection of software components addressing a particular application domain. Thus, we have

- Flexibility in a framework that is achieved through variability in the components and adaptability in the architecture.
- Flexibility in an application that is promoted by making the specific architecture explicit and manipulable.

Development of specific applications is framework-driven, in the sense that all phases of the software lifecycle, including requirements collection and specification are determined according to set patterns formalized within the framework. To a large extent, system design is already done, since the domain and system concepts are specified in the generic architecture. The remaining art is to map the specific requirements to the concepts and components provided by the framework.

[MN95] proposes a scenario where they assume that all parts of the component framework are formally specified, and managed by an application development environment. The environment guides the requirements collection and specification activities, and helps to guide the specialization and configuration of the application from available components.

Given this scenario of component-oriented development, software composition is defined as the systematic construction of software applications from components that implement abstractions pertaining to a particular problem domain. Composition is systematic in that it is supported by a framework, and in the sense that components are designed to be composed.

3.6 Keys of Frameworks

3.6.1 : Analysis, Design and Code Reuse

As it is inferred from the definition and the features described so far, the analysis, design and code reuse is one of the keys of the frameworks. Let's enumerate the reasons given in [FSJ00].

A framework reuses code because it makes it easy to build an application from a library of existing components. These components can be easily used with each other because they all use the interfaces of the framework. A framework also reuses code because a new component can inherit most of its implementation from an abstract superclass. But reuse is best when you don't have to understand the component you are reusing, and inheritance requires a deeper understanding of a class that is using it as a component, so it is better to reuse existing components than to make a new one. Of course, the main reason a framework enables code reuse is that it is a reusable design. It provides reusable abstract design abstract algorithms and a high-level design that decomposes a large system into smaller components and describes the internal interfaces between components. These standard interfaces make it possible to mix and match components and to build a wide variety of systems from a small number of existing components. New components that meet these interfaces will fit into the framework, so component designers also reuse the design of a framework. Finally a framework reuses analysis. It describes the kinds of objects that are important and provides a vocabulary for talking about a problem. An expert in a particular framework sees the world in terms of a framework and will naturally divide it into the same components. Two expert users of the same framework will find it easier to understand each other's designs, since they will come up with similar components and will describe the systems they want to build in similar ways.

Analysis, design and code reuse are all important, though in the long run it is probably the analysis and design reuse that provide the biggest payoff [BR96].

3.6.2 Hotspots and Frozen Spots

The functions in the framework design that need to be customised to change the behaviour of the framework are called "hotspots". Providing a new implementation for such a function makes it possible to add variation to the applications that are based upon the framework. The other functions in the framework are called frozen spots. They represent the static parts of the

framework. These functions define the flow control in the application. They will call the hotspots at the right time.

The template and concrete methods are frozen spots, while the abstract methods are hotspots. The behaviour of template method is inherited by the subclasses of the framework classes. These methods will call at least one template or abstract method that is implemented in the same class or in another class of the framework. The framework developer encodes some of his domain knowledge in the template methods. He knows which steps must be taken to perform one of the framework responsibilities and he transmits this knowledge by calling other methods (abstract as well as concrete) in the template method.

3.7 Analysis: Goals, Benefits and Weaknesses

According to the definition given previously, a framework can be seen as a set of cooperating classes that comprises a reusable foundation for a specific application domain. In this section we enumerate the general features about frameworks considering goals, benefits and weaknesses [FSJ00, Mat96]

3.7.1 Goals

Within the main goals of a framework, we consider the frameworks firstly must minimize the amount of code needed to implement similar applications, since the common abstractions for the applications are captured in the framework, which reduces the fraction of new code to be developed.

Finally, they must optimize generality and leverage. The framework users have to tailor the framework to the applications's specific needs, for example, through subclassing. For leverage, it is necessary that the frameworks include a set of predefined subclasses that the framework user can use at once. Thus if the framework is too general, it will not support the framework user and, if it is not general, it will be useable in very few situations.

3.7.2 Benefits

Talking about benefits, we can mention that the frameworks provide a real decrease in lines of code that have to be developed if the application's required functionality and the functionality captured by the framework are similar to a high degree.

Secondly, an important aspect is that the framework's code is already written and debugged.

Another aspect is that the frameworks offer reuse of design not only code. Framework reusability leverages the domain knowledge and prior effort of experienced developers in order to avoid re-creating and re-validating common solutions to recurring application requirements and software design challenges.

The frameworks also enhance modularity by encapsulating volatile implementation details behind stable interfaces. Framework modularity helps improve software quality by localizing the impact of design and implementation changes

Another benefit is that the extensibility of the frameworks is provided by explicit hook methods [Pre95] that allow applications to extend its stable interfaces. Hook methods systematically decouple the stable interfaces and behaviors of an application domain from the variations required by instantiations of an application in a particular context. Framework extensibility is essential to ensure timely customization of new application services and features.

One of the critical benefits is the run-time architecture of a framework is characterized by an "inversion of control." This architecture enables canonical application processing steps to be customized by event handler objects that are invoked via the framework's reactive dispatching

mechanism. When events occur, the framework's dispatcher reacts by invoking hook methods on pre-registered handler objects, which perform application-specific processing on the events. Inversion of control allows the framework (rather than each application) to determine which set of application-specific methods to invoke in response to external events (such as window messages arriving from end-users or packets arriving on communication ports).

The improved maintenance is another benefit because when an error is corrected in the framework, the same error in software derived from the framework is corrected.

3.7.3 Weaknesses

Taking into account weaknesses, we can say that it is difficult to develop a good framework. Experience in the application domain is necessary when building the framework.

One critical aspect is the documentation of the frameworks because if the frameworks are not supported by the necessary user documentation, they are not likely to be used.

The backward compatibility can be difficult to maintain, since the frameworks evolve and become more mature over time and the applications built on the frameworks must evolve with it.

The debugging process can be complicated because it is difficult to distinguish bugs in the framework from bugs in the application code. If the bugs are in the framework, it can be impossible for the framework user to fix the bug.

Thus it is possible to conclude that the generality and the flexibility of the framework may work against its efficiency in a particular application.

3.8 Summary

This chapter summarizes the main features about object oriented frameworks, and thus, we give the context where we apply the Wright ADL and architectural patterns as complementary techniques to infer an architectural description.

Chapter 4: Architectural Description of Frameworks : First Problems

The terms software architecture and object oriented framework have in common the lack of consensus in their definition. Thus, we presented the most known definitions in the two previous chapters, and we took those that summarize the main features of both concepts. In case of software architecture, we use the classical definition given in [SG96] that defines the software architecture as the structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time. About the definition of object oriented framework, we consider the definition given in [FSJ00] that defines a framework as a reusable design of a system that describes how the system is decomposed into a set of interacting objects. The framework describes both the component objects and how these objects interact. It describes the interface of each object and the flow control between them. It describes how the system's responsibilities are mapped onto its objects.

The most remarkable thing is that both terms seem to be applied in non-intersecting contexts. Only the work proposed by [SG99] shows an approximation of getting an architectural description of frameworks using Wright ADL to represent Sun JavaBeans components. Thus, as our focus is to define a mapping from object oriented frameworks to a software architecture, we must analyze previously which the problems are possible to find and answer them to give a context to our work. This chapter presents some important issues and decisions made during the study of the software architectures and their applicability in object oriented frameworks.

4.1 Software Architecture: Level of Abstraction

As we described in chapter 2, a software architecture is a bridge between domain knowledge, functional and non-functional application requirements, and the design of an application. A stable software architecture needs to retain the concepts and structures from the application domain [Ber98]. On the other hand, successful software architectures often require the 'invention' or discovering of a main mechanism or structure that captures the essence of an application and its solution domains. As such the process of defining a software architecture involves understanding and structuring of domain knowledge, in combination with the creative process of inventing a conceptual model of the application's essence.

An architecture is based on a choice of components and connectors. Components represent a system's main computational elements and data stores: clients, servers, filters, databases, etc. Connections between these elements range from the simple and generic (e.g. procedure call, pipes, shared data access) to the complex and domain-specific (e.g. implicit invocation mechanisms, client-server protocols, database protocols). [AG96] This choice fundamentally determines what the structure will be and what this structure tells us about the system. [Neb98]

There are many valid notions of software architecture that exist at different levels of abstraction. [AG96] says that an architectural description makes a complex system intellectually tractable by characterizing it at a high-level of abstraction. In particular, the architectural design exposes the top-level design decisions and permits a designer to reason about satisfaction of system requirements in terms of assignment of functionality to design elements. The high-level of abstraction is the trend in all the actual literature about software architecture. But in fact the building of an architectural description is a conceptual modeling [Neb98]. A concept is

characterized by its denomination/designation (the names under which the concept is known), its extension (the phenomena described by the concept), and its intension (the set of properties defining the concept). By relating the extension and the intension of an architectural concept we produce new information about the considered software. This is what is commonly known as “moving to another abstraction level”.

Based on this idea, we restrict the use of the term architecture to the semantic structure of an application, a static notion of software architecture, and we consider that what is important is to understand how the classes in a framework are related structurally and what the consequences of this structure are on the software system. We focus our work on a lowest or base-level notion of software architecture that reflects the semantic structure of a software system: the code-level combined with information about design-level in object oriented frameworks. We also adopt a more restrictive definition of architecture as the static instance structure, i.e. the instance structure that is always present during execution and thus characterizes all run-time configuration of instances. Furthermore, the architecture determines which configurations are reachable during execution.

This dissertation has as a main goal the definition of a bridge between the informal nature of the design and implementation process and the formalism required from an architectural description. The proposal works with Wright ADL as the formal language and also the architectural patterns as a complement.

4.2 Architectural Description: Conceptual Model

The previous section gave us the definition and the level of abstraction that we adopt. Based on these concepts, the objective of this dissertation is searching a conceptual model of the classes of the object oriented frameworks, composed of architectural components and connectors, and that represents the semantic structure of an application. We call this model the architectural description of an object oriented framework. Let's see then the main features of the approach and which the level of abstraction is located in.

Some ideas of this dissertation are based in the proposal made by [Now99]. In this work, he proposes the conceptual modelling perspective on software architecture presented in the figure 4.1. The software domain is the reference system (left side of the illustration). The model system (right side of the illustration) is the architectural model which explicitly describes the phenomena and concepts in the software domain from an architectural perspective. During software construction and evolution we form concepts by classification (top left) over the software domain. Examples include design patterns [GHVJ95], architectural patterns [BMR+96], components [Szy98], connectors [SG96] and architectural styles [SG96].

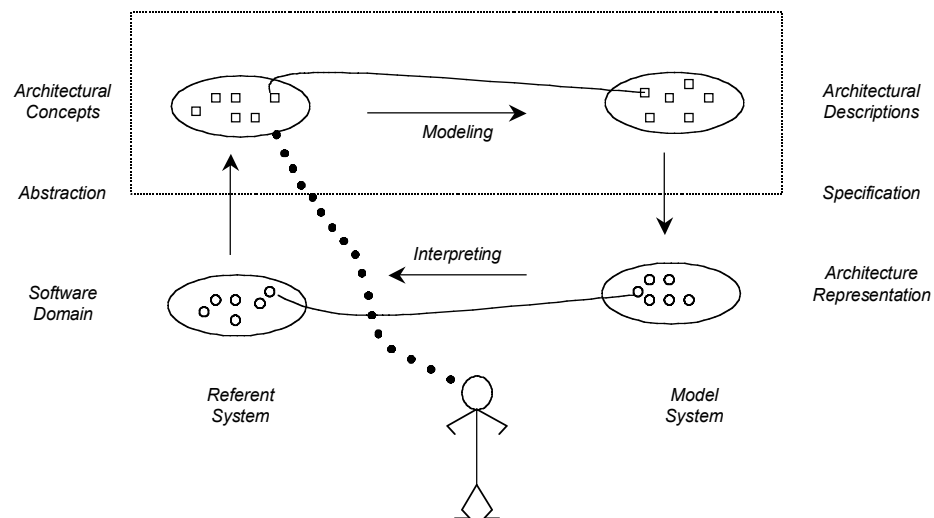


Figure 4.1: Conceptual Modelling of Software Architectures [Now99]

Architectural models seek to describe such concepts in a rigorous way (top right). The descriptions specify representations of architectural phenomena (bottom right), that facilitate the interpretation and understanding of the software domain (bottom-left). But the main problem is that the architectural model is “lost” in the process of modelling an object oriented application (architectural concepts-architectural description in dotted line in figure 4.1) because it is expressed using informal, diagrammatic notations (such as box-line diagrams) and idiomatic characterizations (such as client-server organization, layered system, or blackboard architecture). The meanings of such diagrams and phrases by informal convention are understood only by a small set of developers. This relative informality leads to architectural designs that are inherently ambiguous, difficult to analyze, hard to mechanize.

The objective of “building the bridge” between informal and formal approaches is the definition of a mapping that allows to get an architectural description as an interaction architecture. The concept of interaction architecture is proposed in [Now99]. We want to have the following elements (these are a restricted definition compared to the presented ones in [Now99]):

- ⇒ A *component instance* models a set of collaborating objects providing an interface and a behaviour at execution time. The interface is considered as the service interface and it consists of a set of ports: input ports through signals/information are sent into the component instance and hence affect its behavior; output ports where signals/information are sent out from the component instance as well as affecting other component instances.
- ⇒ A *component class* is a description of a number of similar component instances, and it models a set of object classes describing objects existing before runtime. The objects implement the component instance as described above.
- ⇒ An *interaction instance* models either the creation of an object, the deletion of an object, or the communication of a signal or information between two component instances at runtime. An interaction instance has a role interface consisting of a set of component roles. When connecting interaction instances with component instances we map components to roles. The roles are qualified by a set of required ports.
- ⇒ An *interaction class* is a description of a number of interaction instances with the same component interfaces.

The developed algorithms and the mapping are focused on the definition of component classes and interaction classes mapping from classes in an object oriented framework. The component instances and interaction instances are obtained when the topological configuration is defined using in Wright ADL.

4.3 Architectural Components = Software Components ?

Having decided the level of abstraction of the software architectures where we work and how the architectural elements (components and connectors) will be, two questions should be answered: can we define a direct mapping from software components to architectural components? can the classes in an object oriented framework be considered as software components?. In spite that the relationship between class-object and architectural component appeared in the previous section, and it seems to let us define the direct mapping that we are searching, this section intends to answer these questions and give more elements to work in the context.

Firstly, let's see the concept of software component. [Szy98] summarizes all the definition in the literature considering that a *software component*:

- ⇒ *is a unit of composition and an encapsulated part of a software system*
- ⇒ *has contractually specified interface*
- ⇒ *has context dependencies only*
- ⇒ *serves as the building block for a structure of a system.*

To clarify the concepts, [MN95] explains that the interface of a component defines the component's access points and the access to the services provided by the component. As the component and its clients are developed in mutual ignorance, it is the contract that forms a common ground for successful interaction. The definition requires specification of what the deployment environment will need to provide, such that the components can function. These needs are called context dependencies, referring to the context of composition and deployment. A component's specification of context dependencies must include its required interface and the component world (or worlds) that it has been prepared for. Based on the features given previously, [PW92] proposes two possible categorizations of components:

- Processing elements: supply transformations of the data elements that contain the information that is transformed.
- Data elements
- Connecting elements: which at any time may be either processing elements, data elements or both- constitute the 'glue' that holds the different pieces together.

Another categorization of components developed for the object-oriented programming paradigm is as follows [BMR+96]:

- Controller components
- Coordinator components
- Interface components
- Service provider components
- Information holder components
- Structuring components.

So far, we have obtained the main features of a software components and a two possible categorization of them. Let's see now the definition of object. [Mat00b] defines that an object can be viewed as a unit that holds a cohesive piece of information and that defines a collection of operations (implemented by methods) to manipulate it. Meanwhile, [MKMG96] says that an object oriented design allows system designers to encapsulate data and behaviour in discrete objects that provide explicit interfaces to other objects. They can therefore hide the fact that they might encapsulate existing programs, act as proxies for remote resources, or even coordinate multiple, concurrent requests [MN95].

Specifically, at a programming language level, components may be represented as classes objects or a set of them, because the requirements to be a software component (encapsulated information, specified interface, context dependencies and serve as a building block) are fulfilled by an object according to the definition given previously. Thus, we can provide a direct mapping between a software component (in our case classes) and an architectural component (component

class [Now99]). This process is transparent in the developed mapping. The interface of the architectural components will be the services provided by the classes.

4.4 Connectors

The components does not represent anything by themselves, in fact the most relevant part is the possibility of connecting them providing different types of relationships (defined as connectors in [SG99]). Defining the architectural structure of a framework, another key question is what are the connectors. This question is important because having a clear distinction between the classes, and the mechanisms that coordinates their interaction helps the comprehensibility of the framework. In that way, we isolate two models in the architecture: one for the communication and one for the computation, avoiding to have just only a set of classes without a semantic meaning for the developers. Let's see firstly two ways of identifying possible connectors presented in [SG99] and then study which are the relationships that we can find in an object oriented framework.

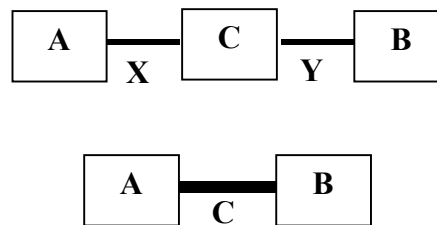


Figure 4.2: Component or Connector ? [SG99]

Consider a system consisting of three components : A, B, and C (figure 4.2). In some cases the purpose of C is to enable the communication between A and B, using A-C specific protocol over connector X, and C-B protocol over connector Y. If those two protocols are completely independent, it makes sense to represent C as a distinct component, and keep X and Y as separate connectors.

On the other hand, if events on X are tightly coupled with those on Y (or vice versa), then it makes more sense to represent the protocol between X and Y directly using a single connector. In this case, the connector itself encapsulates the mediating behaviour of C.

In this dissertation, we take both definitions, because our main goal is the abstraction of the relationships between the classes in the connectors. But in some cases, it is possible to find a class that can be mapped as a connector (second definition). For example, if we can classify classes as connecting, controller, coordinator, interface components, it is obvious that the objective of them is to provide a communication between classes.

Let's enumerate what kinds of relationships (that will map as connectors) we can find in an object oriented framework [BMR+96]. A relationship may be static or dynamic. Static relationships show directly in source code. They deal with the placement of components within an architecture. Dynamic relationships deal with temporal connections and dynamic interaction between components. They may be easily visible from the static structure of source code. Aggregation and inheritance are examples of static relationships. Object creation, communication between objects, and data transfer are usually dynamic relationships. An example of a temporal relationship is when an object is inserted into a container at some point in time and later deleted. Thinking in terms used in an architectural description, [MN95] establishes that a message passing abstraction is used as the glue that connects the objects and defines the communication channels in design component and to describe and encapsulate complex protocols of component interaction that are difficult to describe using traditional object-oriented concepts and notations.

Relationships between components have a great impact on the overall quality of a software architecture. For example, changeability is much better supported by software architectures in which the relationships support the variation of the components, in contrast to

architectures in which any change to a component affects the implementation of its clients and collaborators.

4.5 Internal State of the Components

The most remarkable problem using the Wright ADL is the lack of having internal state of a component. In fact, this ADL is focused on the interaction behaviour and there is no possibility to use the operations given by the components. Most of the executions are hidden behind the non-deterministic choices in the specification. Thus, it is not possible to have any specification about the functional aspects of the components. [San97] proposes the use of Abstract Machines B to solve this problem. Let's see how this mechanism works and how it can be applied in a description made with Wright ADL.

4.5.1 Wright and Abstract Machines B

Wright ADL is focused on the behavior of the interaction and the components themselves are not considered. It is not possible to use the operations which are offered by the components. All the executions specific to the components are hidden in the non-deterministic choices and no specification of the functional aspects of the component is given. It seems interesting to have a specification of the module itself combining Wright with this vision of the module.

To describe a module, the method B [Abr96] is an adequate solution because it allows a checking of the description of the abstract machines and it is able automatically to generate C or Ada code. Details about how the method B works can be found in [Abr96]. But to have an idea of a specification of B, we present the specification of Text_Filter.

An abstract machine B (figure 4.3) has an internal state represented by declarations of variables (line and word for Text_Filter). These variables can be initialized during the first use of the abstract machine. We can declare invariants which must be checked by all the operations.

An abstract machine B provides operations and can import another abstract machine B . Text_Filter has and imports operations to do some tests (IsLetter, IsDigit) or to modify its internal state (BuildWord, IncrementLine). These are the operations that the abstract machine B has and they will be used in this specification.

Knowing how to describe the behavior and the interaction of the components using Wright and the components using the formal specification B, let's see how to join these two concepts respecting their philosophy and their constraints. The idea is to use the method B to describe the components in order to emphasize the internal executions of the components masked in non-deterministic choices.

```

MACHINE
Text
INCLUDES
BASIC_STRING
PROMOTES
IsLetter, IsDigit, IsSeparator, StrComp, Concat, Small
VARIABLES
line
word
INVARIANTS
line  $\hat{a}$  NAT1
word  $\hat{a}$  Word

INITIALISATION
line:=1;
word:=empty

OPERATIONS
bb <- IsEmptyWord = bb:= (StrComp(word, empty)) ;
BuildWord(cc) = PRE cc  $\hat{a}$  Letter THEN word:= Concat(word, cc) END ;
IncrementLine = PRE line + 1  $\hat{a}$  NAT1 THEN line:= line+1 END ;
ww, ii <- Data = ww, ii, word:= Small(word), line, empty

END

```

Figure 4.3: Specification in B of Text_Filter

4.5.2 Interaction of the Abstract Machine B with the environment

The first problem relates to how we express the interaction of the abstract machine B with its environment and then, the connection with the Wright component. Taking into account that a Wright component interacts with his environment by ports, we decide to make the same thing for the abstract machine B by defining a special port called Port B. On the level of the Wright component, we define also a special port called Port Component, which will be used for the interaction with this abstract machine B (Figure 4.4).

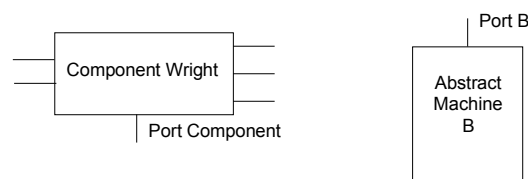


Figure 4.4: Ports Component and B

As the concept of port is not specific to the method B, it is necessary to create an interface which makes the bridge feasible between these two concepts. We can see an abstract machine B as a black box with buttons (operations) which carry out the desired operation pressing them. From this vision of an abstract machine two observations can be considered:

- the interface must contain all the operations offered by the abstract machine B.
- It is necessary to use the deterministic choice of CSP to specify these different operations since the choice of the operation does not depend on the machine but on the environment.

• Thus, if we have an abstract machine B proposing the operations we will have the following specification:

$$Port\ B = (op_1\ ?\ B) _ (op_2\ ?\ B) _ \dots _ (op_n\ ?\ B)$$

A considerable advantage of this specification is that it can be generated in an automatic way. We can now represent our abstract machine B as in the figure 4.5 This interface must thus have a system which allow to link the events op_i with the operations op_i of the abstract machine B. Thus, when the interface receives an event op_i , it will call the operation op_i of the abstract machine.

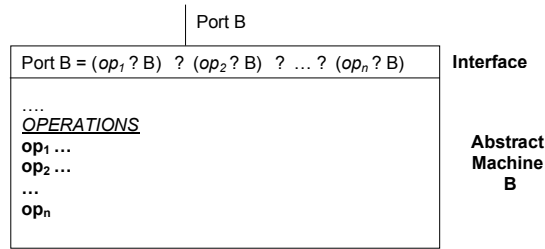


Figure 4.5: An Abstract Machine B with its interface

There is a syntax which links the event and the operation:

- ⇒ The event op_i represents an operation which has not input or output parameters.
- ⇒ The event $op_i?x$ represents an operation which has x as input parameter.
- ⇒ The event $op_i!y$ represents an operation which returns y as output parameter.
- ⇒ The event $op_i?x!y$ represents an operation which has x as input parameter and y as output parameter.

We take the CSP convention that the communications are synchronous and blocking (sending and receiving).

4.5.3 Connection between a Wright Component and an Abstract Machine B

There are two possible solutions to provide the connection between these two ports Component and B (Figure 4.6):

- We can quite simply connect these two ports by a simple link.
- We can make interact these two ports by a special connector called connector B.

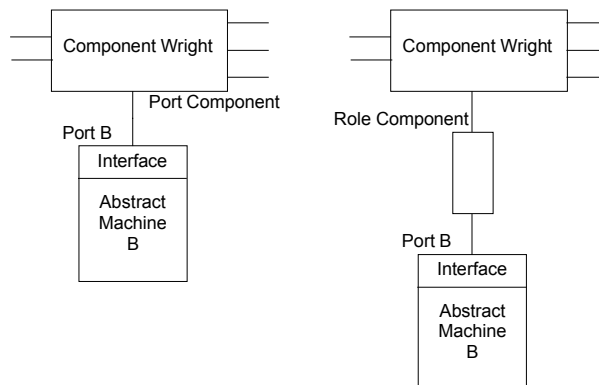


Figure 4.6: Two possible connections between a Wright component and an Abstract Machine B

The second solution is better in this context because:

- The interaction between the Wright component and the abstract machine B requires specifications more complex than those described in the port B.
- By using a connector, we can describe this interaction rigorously while remaining coherent with the Wright component because the specification of Calculation will only describe the interaction of the different ports.
- The use of a connector has the advantage of playing the role of glue between these two concepts of Wright component and abstract machine B. Moreover, this connector makes possible to keep a rather strong independence between these two different concepts.

4.5.4 Description of the interaction between a Wright component and an Abstract Machine B

The goal is to emphasize the internal executions of the component which are masked in non-deterministic choices in the Wright specification. Being specified these executions in the abstract machine B, it is enough to use the port B to reveal them in the Wright specification. Let's see how this is done in the example of the Text_Filter component.

Example: The component Text_Filter receives a set of characters. If the character that read is:

- One letter or one number, it adds the character to the word that is building, and it reads the next character
- The special character B, It increments the line counter and it reads the next character
- One separator, if the previous character was not one separator or this separator is not the first character, it writes the word in small letters and it sends the pair (word, line number) and it reads the next character.

When the component has finished analyzing the set of characters, it sends the last pair and it stops.

Component Text_Filter

Port Input = DataInput

Port Output = DataOutput

Computation =

$$\begin{aligned} & \overline{\text{Input.read}} _ ((\text{Input.data} ?c _ (\text{Computation} \\ & _ \overline{\text{Output.write!}(w,i)} _ \text{Computation}))) \\ & _ (\text{Input.end_of_data} _ \overline{\text{Input.close}} _ ((\overline{\text{Output.close}} _ \$) \\ & _ (\overline{\text{Output.write!}(w,i)} _ \overline{\text{Output.close}} _ \$)))) \end{aligned}$$

Based on this informal specification, it is possible to describe the internal behavior which is masked in the non-deterministic choices. To do that, we use the ports Component and B as well as the connector B. In the connector B we reveal the description of the internal behavior of the component.

On the level of the Wright component, we must describe the interaction between this component Wright and the connector B. This interaction is at the level of the non-deterministic choices:

- In the first non-deterministic choice, where a character is read ($\text{Input.data} ?c$), the internal behavior depends on the read character. Thus we have the new specification:

$$\begin{aligned} & \text{Input.data} ?c _ \overline{\text{Component.write!}c} _ \\ & _ ((\text{Component.continue} _ \text{Computation}) \\ & _ (\text{Component.Data} ?(w,i) _ \overline{\text{Output.write!}(w,i)} _ \text{Computation})) \end{aligned}$$

- For the second non-deterministic choice, where we receive the event *Input.end_of_data*, it should be made sure that there is no data waiting to be sent. The new specification is thus:

$$\overline{\text{Input.end_of_data}} _ \overline{\text{Input.close}} _ \overline{\text{Component.end_of_data}} _ \\ ((\overline{\text{Component.end}} _ \overline{\text{Component.close}} _ \overline{\text{Output.close}} _ \$) \\ _ (\overline{\text{Component.Data?(w,i)}} _ \overline{\text{Component.close}} _ (\overline{\text{Output.write!(w,i)}} _ \overline{\text{Output.close}} _ \$))$$

The new specification of the Texte component is the following one:

Component *Text_Filter*

Port *Input* = *DataInput*

Port *Output* = *DataOutput*

Port *Component* = $(\overline{\text{write!c}} _ (\overline{\text{continue}} _ \overline{\text{Component}} _ \overline{\text{Data?(w,i)}} _ \overline{\text{Component}}))$
 $_ (\overline{\text{end_of_data}} _ (\overline{\text{end_close}} _ \$ _ \overline{\text{Data?(w,i)}} _ \overline{\text{close}} _ \$))$

Computation =

$$\overline{\text{Input.read}} _ ((\overline{\text{Input.data?c}} _ \overline{\text{Component.write!c}} _ \\ ((\overline{\text{Component.continue}} _ \overline{\text{Computation}}) \\ _ (\overline{\text{Component.Data?(w,i)}} _ \overline{\text{Output.write!(w,i)}} _ \overline{\text{Computation}}))) \\ _ (\overline{\text{Input.end_of_data}} _ \overline{\text{Input.close}} _ \overline{\text{Component.end_of_data}} _ ((\\ \overline{\text{Component.end}} _ \overline{\text{Component.close}} _ \overline{\text{Output.close}} _ \$) \\ _ (\overline{\text{Component.Data?(w,i)}} _ \overline{\text{Component.close}} \\ _ (\overline{\text{Output.write!(w,i)}} _ \overline{\text{Output.close}} _ \$))))$$

As we can see, the specification of Calculation only describes the interactions between the different ports. We thus remain coherent with the role of the components given in Wright ADL.

On the level of the connector B, we describe the interaction between the role Component and the role B. Thus, the specification of the B_Texte connector is the following one .

Connector *B_Texte*

Role *Component* =

$$(\overline{\text{write!c}} _ (\overline{\text{continue}} _ \overline{\text{Component}} _ \overline{\text{Data?(w,i)}} _ \overline{\text{Component}})) \\ _ (\overline{\text{end_of_data}} _ (\overline{\text{end_close}} _ \$ _ \overline{\text{Data?(w,i)}} _ \overline{\text{close}} _ \$))$$

Role *B* = $(\overline{\text{IsLetter?c!b}} _ \overline{\text{B}}) _ (\overline{\text{IsDigit?c!b}} _ \overline{\text{B}}) _ (\overline{\text{IsCR?c!b}} _ \overline{\text{B}}) \\ _ (\overline{\text{IsSeparator?c!b}} _ \overline{\text{B}}) _ (\overline{\text{StrComp?(m,w)!b}} _ \overline{\text{B}}) \\ _ (\overline{\text{Concat?(m,c)!w}} _ \overline{\text{B}}) _ (\overline{\text{Small?m!w}} _ \overline{\text{B}}) _ (\overline{\text{IsEmptyWord!b}} _ \overline{\text{B}}) \\ _ (\overline{\text{BuildWord?c}} _ \overline{\text{B}}) _ (\overline{\text{IncrementLine}} _ \overline{\text{B}}) _ (\overline{\text{Data!(w,i)}} _ \overline{\text{B}})$

Glue = $\overline{\text{component.write?c}} _ ((\overline{\text{B.BuildWord!c}} _ \overline{\text{Component.continue}} _ \overline{\text{Glue}}) \\ _ (\overline{\text{B.IncrementLine}} _ \overline{\text{Component.continue}} _ \overline{\text{Glue}}) \\ _ (\overline{\text{B.Data?(w,i)}} _ \overline{\text{Component.continue}} _ \overline{\text{Glue}}) \\ _ \overline{\text{Component.continue}} _ \overline{\text{Glue}}) \\ _ \\ (\overline{\text{Component.end_of_data}} _ ((\overline{\text{Component.end}} _ \overline{\text{Component.close}} _ \$)$

$$\overline{(B.Data?(w,i) _ Component.Data!(w,i) _ Component.close _ \$))}$$

In this specification of the connector B, we see the description of the internal behavior of the component using the events of the role B. However, it is important to note that our special connector B is not completely a Wright connector. The connector B_Texte makes internal choices which are represented by non-deterministic meanwhile a Wright component can not make internal choices and only describes the interaction between different components. However, this is not incoherent with the idea of connector in our case. Starting we know that the special connector B did not have same specificity comparing Wright connectors. There are two reasons:

- A Wright connector can have different styles of interactions: connector pipe-filter, client-server, whereas the purpose of our special connector B is precisely to describe only one interaction: internal behavior of the component.
- The purpose of a Wright connector is only to describe the interaction between various components. The connector B does not aim to connect various components but rather to describe the interaction between the Wright component and its abstract machine B which represents in fact one component.

4.5.4.1 Resolution of non-determinism in the connector B

The non-determinism of the Wright component which is in the connector B_Text can be solved. Thanks to the abstract specification of the component Text_Filter, we know to clarify these non-deterministic choices. In our case, the non-deterministic choices correspond to tests of the if-then-else type. In the figure, a little abstract specification of this resolution of the non-deterministic choices is given. There is a notation in CSP to express the test of the if-then-else [Hoa85]: the process $P < |b| > Q$ behaves as the process P if the Boolean expression B is true and like the process Q if not.

Thus we can describe in a formal way the connector B_Texte specified in an abstract way.

Connector B_Texte

$$\text{Role Component} = \overline{\text{write!c}} _ \overline{\text{continue _ Component _ Data ?(w,i) _ Component}} _ \overline{\text{end _ of _ data _ (end _ close _ \$ _ Data ?(w,i) _ close _ \$)}}$$

$$\text{Role B} = \overline{\text{IsLetter?c!b _ B}} _ \overline{\text{IsDigit?c!b _ B}} _ \overline{\text{IsCR?c!b _ B}} _ \overline{\text{IsSeparator?c!b _ B}} _ \overline{\text{StrComp?(m,w)!b _ B}} _ \overline{\text{Concat?(m,c)!w _ B}} _ \overline{\text{Small?m!w _ B}} _ \overline{\text{IsEmptyWord!b _ B}} _ \overline{\text{BuildWord?c _ B}} _ \overline{\text{IncrementLine _ B}} _ \overline{\text{Data!(w,i) _ B}}$$

Glue =

component.write?c _

if (B.IsLetter!c?b) or (B.IsDigit!c?b) then (_ B.BuildWord!c _ _ Component.continue _ Glue)

else if (B.IsCR!c?b) then (_ B.IncrementLine _ _ Component.continue _ Glue)

else if (B.IsSeparator!c?b) and not (B.IsEmptyWord?b)

then (_ B.Data?(w,i) _ _ Component.continue _ Glue)

else _ Component.continue _ Glue)

_ (Component.end_of_data _ if (B.IsEmptyWord?b) ((_ Component.end _ _ Component.close _ \\$)

else (_ B.Data?(w,i) _ _ Component.Data!(w,i) _ _ Component.close _ \\$)))

We could also choose to carry out these tests by the abstract machine B which is more suitable to describe tests than CSP. The problem is that it is not very easy to specify the behavior

which must follow the Wright component. We could, indeed, describe an operation in B which makes all the tests and calls the corresponding operations but then it is necessary to be able to specify when the B_Texte connector must initialize the event $\overline{Component.continue}$ or the event $\overline{Component.Data!(w,i)}$ or the event $\overline{Component.end}$.

4.6 Summary

This chapter summarizes the main specific difficulties that we found when we started to study the applicability of the concepts of software architecture and ADLs to describe an object oriented framework. We consider important to enumerate them because they represent the basis for the developed approaches of this dissertation.

Chapter 5: Architectural Description OF Object Oriented Frameworks: An Approach

The overall design and the specification of an object oriented framework using box-line drawings artefacts usually makes us lose the semantics of the main architectural concepts that were thought when the framework was developed. Besides this, we can not have any information such as the points of variability, reuse, if it can be integrated to other ones or a reference to measure the changes in subsequent versions. The ADLs can be the answer to these issues. Thus, in this chapter, we present the approach to cope with the problem of a lack of a semantic structure of an object oriented framework: Two algorithms to get an architectural description of frameworks using as tools Wright ADL and architectural patterns. As the formal basis of Wright ADL is CSP [Hoa85], we also define a mapping from Smalltalk/Java code to CSP notation. We use all the analyzed points presented in the previous chapter.

Our approach is an alternative one to the classical proposal of working with software architectures. In fact, the common applicability of the concept of software architectures is the inference of an architecture of pre-built systems in a 'high level' of abstraction. Mostly, these systems were not designed using the object oriented paradigm, and thus, we do not have any information about rules or mechanisms of how to discover components and interrelationships. The main idea of this proposal is to provide a way to do it in object oriented frameworks.

5.1 Introduction

There is very few work made in software architectures applied in object oriented applications. It is possible to see all the work made in terms of (design) patterns to make the object oriented applications easier to understand talking about 'similar' models. For example, if you find in the design that the classes A, B and C were modelled using the Composite pattern [GHVJ95], the user associates a picture such as shown in figure 5.1 and clearly knows the way of working between the different classes.

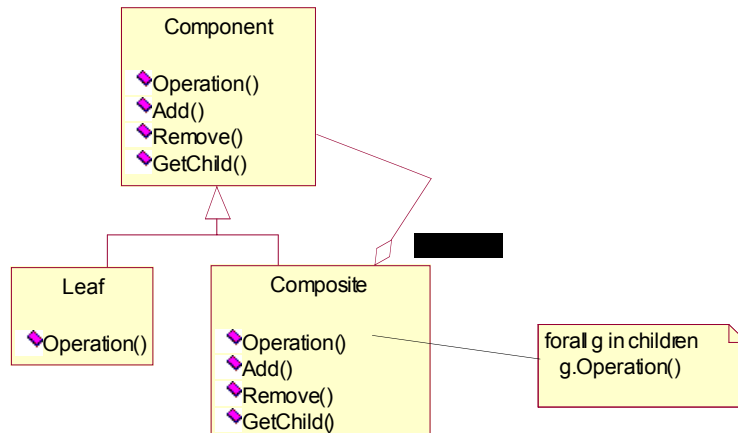


Figure 5.1: Composite Pattern

But not all the applications can be seen with known structures such as design patterns [GHVJ95]. In fact, all the details about an application in general should be provided in the documentation of it. However, one of the main problems in frameworks is the lack of clear documentation or a better complement between the different methodologies/techniques to understand functionality and structure and also fulfill needs depending on the kind of users of frameworks. Usually, all the documentation of the process of building a framework or of the framework itself is poor and when the developer has finished the framework the possibility of understanding how the different classes of the framework are connected or what their collaborations are to perform a task is a 'titanic' work. The classes diagrams show design patterns or known structures of classes considering different relationships between two (set of) classes with a line. For example, in the figure 5.2 different generic relationships are shown: composition (diamond), hierarchy (triangle), creation (dotted line) and generic relationship (full line). But if you want to know exactly what the line means or how the composition between two classes is used (share a data, synchronic communication), you have to search for collaboration diagrams, for example, or in the worst case, have a look at the code.

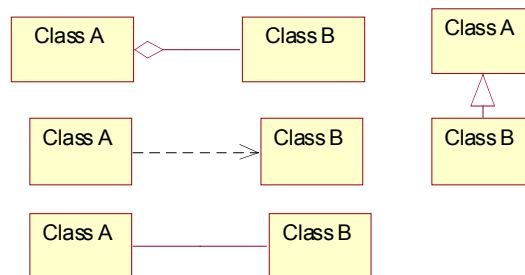


Figure 5.2: Graphical Relationships between classes

The ADL seems to be one possible answer to this lack of semantics in terms of interaction protocol between the classes. In this chapter, we consider two approaches to get an architectural description of object oriented frameworks. Both approaches were thought as algorithms (a set of steps). In both cases, we work with frameworks in which all the design decisions were already made. The first approach defines a set of steps to get an architectural description of a framework, based on that the documentation of the framework is the design expressed with class hierarchies using UML notation [UML], notes about the design and the code itself. Using these three artefacts, the algorithm infers an architecture of the framework in terms of components and connectors explicitly. The second approach is a derived algorithm from the first one. The algorithm defines a set of steps to get also an architectural description of an object oriented framework

component, based on that the documentation of the framework component is only the design expressed with the class hierarchies in UML notation [UML] and notes about the design. In the second approach we work with a part of a framework (a component) and the main idea is the use of ADLs as one description tool to help us in the process from the design to the implementation, expressing the implementation in terms of interaction protocols and in the code at the same time. The main difference between two algorithms is that in the first one we already have the framework implemented, and in the second case, we use the algorithm to help in the implementation.

We use Wright ADL to represent all the relationships between the classes in a framework. Before starting to define how we get an architectural description in terms of components and connectors, we present a defined mapping between code written in Java or in Smalltalk in terms of CSP process. This lets us define easily the protocol for the ports and computation of the components and the role and glue of the connectors

5.2 MAPPINGS: Definition and Assumptions

A description of the different elements of Wright ADL was presented in the chapter 2. Briefly, in this ADL, a **component** is defined by an **interface** and a **computation**. The interface consists of a number of **ports**. Each port represents an interaction in which the component may participate. A **connector** consists of a set of connector **roles** and the connector **glue**. Each role specifies the behavior of participant in the interaction and the glue describes how the participants work together to create an interaction. Thus, the general structure of a component and a connector in Wright ADL is the following one:

```
Component ComponentName
  Port NamePort1 = ...
  ...
  Port NamePortn = ...
  Computation = ...
```

```
Connector ConnectorName
  Role NameRole1 = ...
  ...
  Role NameRolen = ...
  Glue = ...
```

As our first objective of working in different levels of description is to be closer to the code going from one coarse-grained level to fine-grained level of description, firstly we show the mapping from the code to the architectural elements. Our mapping must be in terms of CSP process to be able to use it in the Wright ADL. It must be clear enough to the reader that in most of the cases, we are constrained by the possibilities to represent architectural elements with Wright ADL.

5.2.1 Mapping for Classes

A direct mapping between classes and components is made. This means that each class in the class model is considered as a component in the description. But we decouple all the information about class communication between the component and the connector. All the information about the communication to other objects is put in the connector. For example, supposing that we have a class model to represent a book that is composed pages (See Figure 5.3).

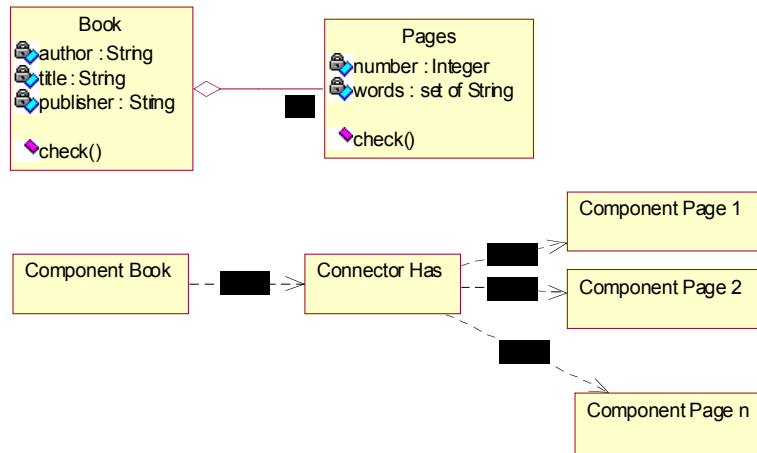


Figure 5.3: Class Model for a Book and the Mapping with Connectors and Components

These classes are mapped as two components: Book and Page, and we get a connector Has which represents the relationship between the two components. The behaviour of the component regarding to the other objects is left to the connector. In this case, for example, when the Book wants to do a spelling checking, it just only sends the event to the connector which forwards the events to the pages. This fact is also illustrated graphically in Figure 5.3.

5.2.2 Mapping for Relationships between Classes

Besides of the relationships that can be mapped by the message sendings (explained in the next subsection), we also consider three kinds of relationships between two classes A and B.

- instances of B can be instances/class variables of A
- instances of B used as parameters in one method of A
- instances of B are connected to an instance of A by a dependency mechanism.

In all the cases, we represent A, the instances/class variables of A and the parameters as components in the Wright description whenever the objects are not instances of primitive classes (in Smalltalk) or primitive types (in Java) such as Char, Integer, Boolean. We only want to keep objects with a composite structure. For example, in the figure 5.4 we can see the class model of a Truck and its representation in terms of components and connectors. It must be clear for the user that this is a complete representation for this model., this means that we can simply represent the Truck and avoid any information of the Manufacturer. In this case we represent them because we use the information of the latter. But it is clear that we do not have, for example, a component for the name of the Manufacturer. All the information (called as simple) can be used as parameters. This assumption is taken because the management of parameters in Wright ADL is limited to simple parameters such as letters and integers.

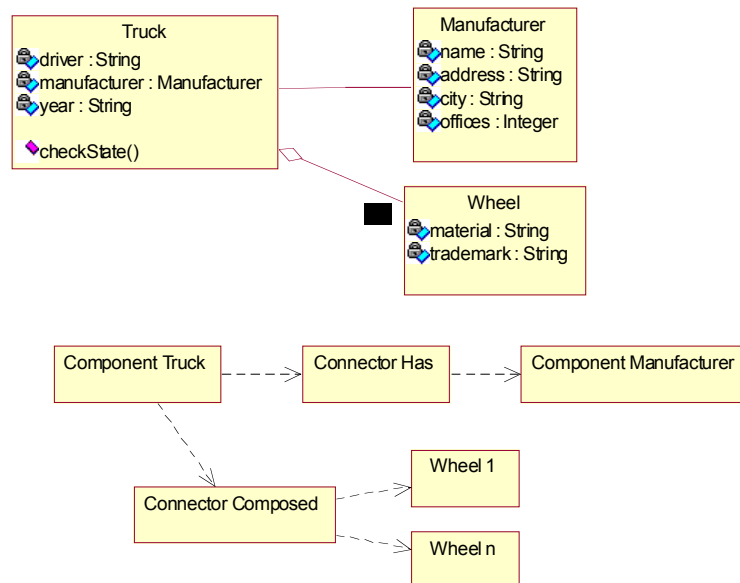


Figure 5.4: Class Model for a Truck and Representation with Components and Connectors

5.2.3 Format for Components and Connectors

As our approach focuses on having components only as units of computation and connectors as units of communication, we model the classes as components without any knowledge of what objects are connected with and we leave all this information to the connectors. In this way, the names of the ports in the component are left to the user, but we adopt the names of the roles in the connector with the names of classes that must be connected. If an instance of class A must connect with instances of class B, the connector have the following format:

Connector AB
Role A = ...
Role B_{1..n} = ...
Glue = ...

5.2.4 Mapping for the Messages

The method calls in a method m_k in a class A have the following formats:

- in Smalltalk:
 - $object_i message_j$
 - $object_i message_j: p_1 with: p_2 ... with: p_n$
- in Java:
 - $object_i . message_j$
 - $object_i . message_j (p_1, p_2, ... p_n)$

where $object_i, p_1, p_2, ... p_n$ can be instances/class variables of A, parameters in the method or the class itself and $message_j$ can be seen as a service that the class initializes or simply the notification of a change (dependency mechanism). We are assuming that objects p_i have only one level of objects' composition in their structure. Following with the example of the Truck (Figure 5.4), p_1 can be an instance of the Wheel class, but it can not be an instance of the Truck Class, because this latter has two levels of composition. We are making this assumption because we are

interested in being able to decompose the parameter p_i in terms of its components $p_x(o_{x1}, \dots, o_{xm})$. In the case of *Manufacturer*, we will get *manufacturer(name, address, city, offices)*. Thus, we get a good level of expressiveness in the description. Based on the structure of methods, let's see the different focus that we have about the methods themselves and their bodies. Firstly, all the methods (m_k) are mapped as events in CSP. Thus, we must know if the methods are called by the another object or if the method is a 'shooter' of actions. In the first case, the method will be mapped as an **observing event** $\overline{m_k}$, and in the second case, the method will be mapped as an **initiating event** m_k . So the component A (class A mapped as component in the description) has the following structure:

Component A

$$\begin{aligned} \mathbf{Port\ Out} &= (m_k \mid \overline{m_k}) \mid \\ & (m_k \mid \overline{m_k} _ \overline{message_j!objectName} _ \forall x \lfloor 1..n; \overline{asParameter!p_x(o_{x1}, \dots, o_{xm})}) \\ \mathbf{Computation} &= (\overline{Out.m_k} \mid \overline{Out.m_k}) \mid \\ & (\overline{Out.m_k} \mid \overline{Out.m_k} _ \overline{Out.message_j!objectName} \\ & _ \forall x \lfloor 1..n; \overline{Out.asParameter!p_x(o_{x1}, \dots, o_{xm})}) \end{aligned}$$

Connector AB

$$\begin{aligned} \mathbf{Role\ A} &= (m_k \mid \overline{m_k}) \mid \\ & (m_k \mid \overline{m_k} _ \overline{message_j!objectName} _ \forall x \lfloor 1..n; \overline{asParameter!p_x(o_{x1}, \dots, o_{xm})}) \\ \mathbf{Role\ objectName} &= \overline{message_j} _ \forall x \lfloor 1..n; \overline{asParameter?p_x(o_{x1}, \dots, o_{xm})} \\ \mathbf{Glue} &= (\overline{A.m_k} \mid A.m_k) _ \overline{A.message_j?objectName} \\ & _ \forall x \lfloor 1..n; \overline{A.asParameter?p_x(o_{x1}, \dots, o_{xm})} _ \overline{objectName.message_j} \\ & _ \forall x \lfloor 1..n; \overline{A.asParameter!p_x(o_{x1}, \dots, o_{xm})} \end{aligned}$$

The connector uses a name matching with the parameter *objectName* and thus identify to which component (identified with the portname) it sends the events.

5.2.5 Mapping for Classes' Creation

Class A must create instances of classes B in one of its methods, so we find the following codeline:

- in Smalltalk:
 - *B new*
 - *B new: p₁ with: p₂ ... with: p_n*
- in Java:
 - *B ()*
 - *B (p₁, p₂, ..., p_n)*

In the first case, it maps as a special event name $\overline{create!B}$ in the protocol of the component. The class A is creating an element B so it is an initiating event. In the second case, it maps as a sequence of events $\overline{create!B} _ \forall x \lfloor 1..n; \overline{asParameter!p_x(o_{x1}, \dots, o_{xm})}$

5.2.6 Mapping for Conditional Statements

In the methods calls we can also have a conditional statements.

- in Smalltalk:
 - (condition) ifTrue: [actionTrue]
 - (condition) ifFalse: [actionFalse]
 - (condition) ifTrue: [actionTrue] ifFalse: [actionFalse]
 - (condition) ifFalse: [actionFalse] ifTrue: [actionTrue]
 - (condition) whileTrue: [action]
 - (condition) whileFalse: [action]
 - 1 to: n do: [action]
- in Java:
 - if (condition) actionTrue
 - if (condition) actionTrue else actionFalse
 - while (condition) action
 - for (i:=0; i++; i<=n) action

where the condition can only be one boolean expression (e_1) or a set of boolean expressions (e_1, \dots, e_n) joined by logic operators (*and*, *or*, *xor*), and the *action*, *actionTrue* and *actionFalse* can be a method call (m_1) or a sequence of method calls (m_1, \dots, m_k) (as we explained previously). In the case of expressions, they are method calls which reply True or False. Thus in both cases, we consider them as events inside the description.

Firstly, we study the condition. The expressions e_1, \dots, e_n are a sequence of method calls (except in the case they evaluate an internal state of the object, e.g. comparing two values of instances variables), so we map the condition such as:

$$\text{Process Condition} = \forall x \lfloor 1..n; \overline{e_x} _ (\text{answer?True} _ \dots _ \text{answer?False} _ \dots)$$

Thus we leave the responsibility of evaluating the logic expression to the connector. Let's see how the process in the connector would be modeled.

Connector Logic (nb: $1..n$)

Port A = Condition

Port B $1..nb = e_j _ (\overline{\text{answer!True}} _ B _ \overline{\text{answer!False}} _ B)$

Computation = $\forall x \lfloor 1..n; (A.e_x _ B_x.e)$

if the logic operator is 'and' : $\forall x \lfloor 1..n; B_x.\text{answer?True} _ \overline{A.\text{answer!True}}$
 $_ \exists x \lfloor 1..n; B_x.\text{answer?False} _ \overline{A.\text{answer!False}}$

if the logic operator is 'or' : $\exists x \lfloor 1..n; B_x.\text{answer?True} _ \overline{A.\text{answer!True}}$
 $_ \forall x \lfloor 1..n; B_x.\text{answer?False} _ \overline{A.\text{answer!False}}$

We consider the possibility of expressing explicitly the condition. But if the user decides to avoid it, the events to execute when the condition is true or false are expressed as non deterministic choice because we do not have any information about the condition, so we can think that the component takes the decision of the actions to follow. In the other case, we must use the deterministic choice, because it has an observing event (*answer*) which communicates the result of the condition. Let's see how it works.

The if-statement maps as a non deterministic choice with the following format:

$$\text{Process if} = \overline{\text{actionTrue}} _ (\text{or } \forall x \lfloor 1..n; \overline{\text{actionTrue}}_x)$$

$$_ \overline{\text{actionFalse}} _ (\text{or } \forall x \lfloor 1..n; \overline{\text{actionFalse}}_x)$$

But in the case of using the condition explicitly:

$$\text{Process } \mathbf{if} = \text{all the conditions are sent } _ \\ (\text{answer?True } _ \overline{\text{actionTrue}} \text{ (or } \forall x \lfloor 1..n; \overline{\text{actionTrue}}_x \text{) } \\ _ \text{answer?False } _ \overline{\text{actionFalse}} \text{ (or } \forall x \lfloor 1..n; \overline{\text{actionFalse}}_x \text{)})$$

In the case of WHILE-statement, using non-deterministic choice:

$$\text{Process } \mathbf{While} = \overline{\text{action}} \text{ (or } \forall x \lfloor 1..n; \overline{\text{action}}_x \text{) } _ \mathbf{While} _ \S$$

But using a deterministic choice:

$$\text{Process } \mathbf{While} = \text{all the conditions are sent } _ \\ (\text{answer?True } _ \overline{\text{action}} \text{ (or } \forall x \lfloor 1..n; \overline{\text{action}}_x \text{) } _ \mathbf{While} \\ _ \text{answer?False } _ \S \text{)}$$

The For-statement maps as a non-deterministic choice with the following format:

$$\text{Process } \mathbf{For} = \overline{\text{action}} \text{ (or } \forall x \lfloor 1..n; \overline{\text{action}}_x \text{) } _ \mathbf{For} _ \S$$

But using a deterministic choice:

$$\text{Process } \mathbf{For} = \text{execute}_0 \\ \text{where } \text{execute}_i = \overline{\text{action}} \text{ (or } \forall x \lfloor 1..n; \overline{\text{action}}_x \text{) } _ \text{execute}_{i+1} \text{) } \mathbf{when } i \text{ in } 1..n \\ \S \text{ otherwise.}$$

5.3 Object oriented Architectural Description - First Case: Design and Code

Based on different object oriented application frameworks we have developed an algorithm (set of steps) to infer a possible architecture which represents the set of classes and different relationships between them. The main idea is just to infer a set of components and connectors which shows the complete behaviour of an application framework. We show two approaches:

1. Use of Predefined Style: components and connectors defined in other configurations
2. Use of Steps: Start from the « scratch » following closer to the design and implementation decisions made in the framework

Firstly, we explain why we decided not to take the first approach as our main approach. We use an example for both cases, because it will turn the process more illustrative and understandable.

5.3.1 Example: Measurements System Framework [Bos00]

Measurement systems are a class of systems to measure the relevant values of a process or product. It is used for quality control of parts entering production or of produced products that can then be used to separate acceptable from unacceptable items to categorize the products in quality categories.

A measurement system consists of more than sensors and actuators. A typical measurement cycle starts with a trigger, indicating that a product, or a measurement item, is

entering the system. The first step after the trigger is the data collection phase performed by the sensors. The sensors measure the relevant variables of the measurement item. The second step is the analysis phase, during which the data from the sensors are collected in a central representation and transformed until they appear in a form in which they can be compared to the ideal values. Based on this comparison, some discrepancies can be deduced, which in turn, lead to a classification of the measurement item and is used to perform associated actions, such as rejecting the item, which causes the actuators to remove the item from the conveyer belt and put it in a separate store, or to print the classification on the item so that it can be automatically recognized at a larger stage. One of the requirements for the analysis phase is that the way the transformation takes place, the characteristics on which each classification should be flexible and easily adaptable during system construction, but also, to some extent during the actual system operation.

5.3.1.1 Architecture of the System

[Bos00] identifies five entities that communicate with each other to achieve the required functionality.

1. The trigger triggers the abstract factory when a physical item enters the system
2. The abstract factory creates representation of the physical object in the software, that is, the measurement item.
3. The measurement item requests the sensor to measure the physical object.
4. The sensor sends back the result to the measurement item that stores the results.
5. After collecting the required data, the measurement item compares the measured values with the ideal values.
6. The measurement item sends a message to the actuator requesting the actuation appropriate for the measured data.

The proposed architecture is shown in the Figure 5.5. For the approach, we work only with the microarchitecture composed by ItemFactory, Measurement Item and Actuator. All the architectural description complemented with design techniques such as collaboration diagrams and class hierarchies can be consulted in Appendix A.

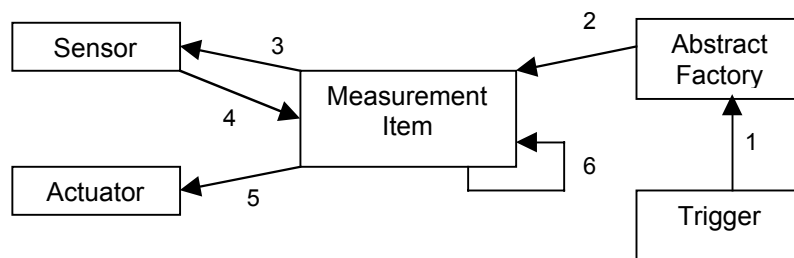


Figure 5.5: Architecture of the Measurement System Frameworks

5.3.1.2 Minimal details about the design

The Item Factory incorporates both the Abstract Factory and the Prototype design patterns [GHVJ95] and it is responsible for instantiating instances of class Measurement Item whenever it receives a trigger event, to configure these instances and to activate each instance by providing it with a separate process or invoking its start method. The Item Factory class contains an instance of Measurement Item denoted as prototype item and a state variable, inCalibration, indicating whether the system is in calibration mode or in normal operation.

This system can be in two possible states: Normal or Calibration state. In the Normal state, the system works exactly as it was described in the steps previously. But if it is in Calibration state, the Item Factory and the Measurement Item both play a role. The Item Factory contains an instance of Class Measurement Item that it is used as a prototype for generating measurement items during normal system operation. When the user of the system decides to calibrate, the first step is to notify the item Factory that the next measurement item is to be used for calibration. When the trigger notifies the item factory, the item factory instantiates a measurement item, but replaces the normal actuation strategy with a calibration strategy. This causes the item factory to replace its current prototype item with the last measurement item. Now, all following trigger events will result, in that the item factory instantiates a copy of the measurement item now stored as the prototype item; in other words, the prototype measurement item contains the new set values.

The measurement item is the object that contains the data collected from the sensors concerning the physical measurement item. It is invoked via its start method by the item factory after it has instantiated the item. The start method contains the top-level behavior for the measurement item with respect to its primary task – to collect data on the physical item it represents and to actuate the actuators appropriately, based on the measured data and the comparison to the set data.

The calculation phase of the measurement item is concerned with collecting the data and converting it to a value form that matches the requirements within the system. The only action performed by the measurement item is to invoke each measurement value that it contains with a request to collect the data from the sensor to which the measurement value is connected and to process this data. The actuation phase is concerned with generating the necessary effects on the actuators, based on the values collected during the calculation phase. In the case of Calibration state, the actuators are not considered and the actuation strategy provokes that the item factory makes a new prototype using the last measurement item as a basis.

5.3.2 First Approach: Use of Predefined Styles

The first approach is based on the use of predefined styles given in other systems' description using Wright ADL].

Supposing that we identify the classes Item Factory and Measurement Item as components, and we analyze the behaviour between these two classes. We can consider that the behaviour can be mapped as a pseudo Master/Slave style, where the master is the Item Factory and the slave is the Measurement Item. The item factory creates objects to measure specific values on a physical item, and when the task is done, and if the item factory has not to copy the last measurements as the new prototype, it destroys the object. Behind these conditions, we establish the relationship between them in terms of master/slave style and we used a predefined style (shown in the figure 5.6).

Interface Type *SlavePullT* = $\overline{\text{open}} _ \text{Operate} _ \$$

where **Operate** = $\overline{\text{request}} _ \text{result?x} _ \text{Operate} _ \text{close} _ \$$

Interface Type *MasterPushT* = $\text{open} _ \text{Operate} _ \$$

where **Operate** = $\text{request} _ \overline{\text{result!x}} _ \text{Operate} _ \text{close} _ \$$

Connector *MasterSlave*

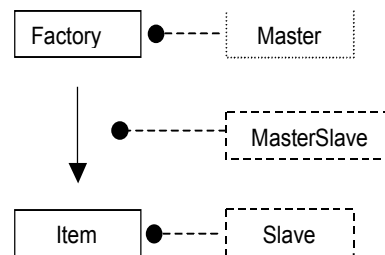
Role *Slave* = *SlavePullT*

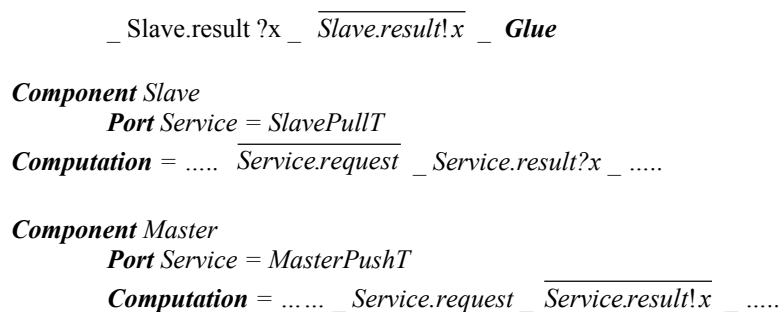
Role *Master* = *MasterPushT*

Glue = *Slave.open* $_ \overline{\text{Master.open}}$ $_ \text{Glue}$

$_ \text{Slave.close}$ $_ \overline{\text{Master.close}}$ $_ \text{Glue}$

$_ \text{Slave.request}$ $_ \overline{\text{Master.request}}$ $_ \text{Glue}$



**Figure 5.6: Master / Slave Pattern**

Let's see then what the advantages and disadvantages are using this approach. The use of a predefined style gives us a generic and global overview of the system and also we have the reuse of components and connectors already defined in other architectures. But, in this case we have to study the adaptability of other configuration to our system and, as the use of ADLs to represent architectures is relatively new and is not often used for object oriented applications, there is no catalog (such as design patterns[GHVJ95]) of predefined components and connectors. Finally, in spite of the generic view that we get, this way of description does not give us enough information about the connection between the components in terms of messages between objects. This means that if we are interested in more details closer to the implementation, we need to have a meaning for the different events in the architecture. For example, is the event *open* a message between the two classes or just a way of representing a set of messages between the two classes ?. As our approach focuses on mapping messages in terms of events, we decided not to take this approach. We are not claiming that it is an useless approach, but according to our needs it does not fulfill our requirements.

5.3.3 Second Approach: Use of Steps

Having as documentation the design expressed in class hierarchies and the code itself, this set of steps for describing frameworks has the objective of working in different levels going from the domain-specific to implementation-specific issues. We follow the proposed microarchitecture to understand the applicability of the different steps.

Step 1: Identify the main classes of framework in terms of the domain. This step is concerned with identifying classes which were mapped of concepts of the studied domain where the framework will be applied. In most of the cases these classes are clearly identified in the design. In the case of having classes hierarchies, we suggest to take the root class of the hierarchy. For example, considering our micro-architecture of the MSF, we have three main classes: Actuator, Item Factory and Measurement Item.

Step 2: Each class is mapped to a component and each possible relationship between two classes is mapped to a connector in terms of Wright ADL, avoiding to have a relationship with classes of simple types (integer, char, boolean). This step is concerned with getting relationship between classes which are composed of other objects. In our example, we define four components: Actuator, Item Factory and Measurement Item mapped from the classes and Measurement Value mapped from the composition relationship with Measurement Item Class.

Step 3: The protocols of each classes are classified as initiating or observing events, and all the messages called in the body of the messages are classified as initiating events. We avoid to take into account methods classified in protocols categories such as initializing or accessing and also the assignments in the implementation. We explain later the reasons and the limitations of Wright ADL.

The Component ItemFactory and the Component Measurement Item has three observing events that work together: trigger/start to indicate the beginning of its working; and normal and calibrate to indicate the state in which the system is. In the case of Component ItemFactory they are also initiating events because they are used to initialize the Measurement Item. The Measurement Item has two initiating events: performCalculation to make the values measure and calibrate to indicate the calibration process to the values. The Component Measurement Value and the Component Actuator (in this microarchitecture) are only receivers of events of the Measurement Item. Thus all their events are observing ones.

Step 4: The protocols for the ports and the computation of the components are built.

Class ItemFactory

iv: prototypeltem inCalibration

trigger

“called by the trigger sensor to indicate that a measurement item has entered the system”

| mi as |

mi:=prototypeltem copy.

(inCalibration)

ifTrue:[as:=CalibrationStrategy new. inCalibration:=false]

ifFalse:[as:=ActuationStrategy new].

as context: mi.

mi actuationStrategy: as.

mi factory: self.

mi start.

^mi

calibrate

“sets the calibration state so that the next measurement item is used for calibration”

inCalibration:=true.

^self

calibrate:aMeasurementItem

“makes the argument the prototype item”

prototypeltem:= aMeasurementItem

^self

Component Item_Factory

Port In = $\overline{\text{trigger}} \rightarrow \text{In}$ $\overline{\text{calibrate}} \rightarrow \text{\$}$

Port Out = $\overline{\text{start}} \rightarrow (\overline{\text{calibrate}} \rightarrow \text{Out} _ \overline{\text{normal}} \rightarrow \text{Out})$

Computation = $\text{In.trigger} \rightarrow \overline{\text{Out.start}} \rightarrow$
 $(\overline{\text{Out.calibrate}} \rightarrow \text{Computation}$
 $_ \overline{\text{Out.normal}} \rightarrow \text{Computation})$
 $_ \text{In.calibrate} \rightarrow \text{Computation}$
 $_ \text{\$}$

The Component Item_Factory has one observing event trigger which is activated by the Class Trigger. Then it will start an measurement item in two possible states: normal or calibration. As this condition is based on an internal state of the component, we used a non-deterministic choice. In the method start, this condition is changed by assigning different algorithms to ActuateStrategy. We did not find another adequate representation and we decided to express as two possible states of starting the process.

Class MeasurementItem

iv: sensor, measurementValues, actuators, myItemFactory actuationStrategy

performCalculation
 measurementValues do: [:n | n performCalculation]
 ^self

calibrate
 “perform calibration”
 measurementValues do: [:mv | mv calibrate]

start
 (self calculationStrategy) performCalculation.
 (self actuationStrategy) actuate
 ^self

Component Measurement_Item

Port In = start → (normal → In_calibrate → In)

Port Out = $\overline{\text{performCalculation}} \rightarrow \text{Out_calibrate} \rightarrow \text{Out}$
 $\overline{\text{actuate}} \rightarrow \text{Out}$

Computation = In.start →
 $(\overline{\text{In.calibrate}} \rightarrow \overline{\text{Out.performCalculation}} \rightarrow \overline{\text{Out.calibrate}})$
 $\text{In.normal} \rightarrow \overline{\text{Out.performCalculation}} \rightarrow \overline{\text{Out.actuate}}$)
 → **Computation** _ \$

The component protocol is very simple. Once the measurement item is created in both states, just ask for the measurements to the measurements values.

Class MeasurementValue

performCalculation

calibrate

Component C_MeasurementValue

Port In = performCalculation → In_calibrate → In

Port Out = ...

Computation = In.perform_calculation
 → “call the sensors” → **Computation**
 $\overline{\text{In.calibrate}}$
 → “calibrate their values” → **Computation**
 _ \$

The Measurement Value and Actuator are also simple because just only receives the requests from the Measurement Item and executes their requests.

Class Actuator

actuate
 “Perform the actuation associated with this actuator”

inActuation value:true.
 (hardwareActuator isNil) iffFalse: [hardwareActuator actuate].
 InActuation value:false.
 ^self

Component *Actuator*

Port *In* = *actuate* → *In*

Computation = *In.actuate* → “call the actuators” → ... → **Computation**

Step 5: The connectors are built using the messages sent from one class to another one. In this case, we have three connectors *Item_MI*, *MI_Act* and *MI_MV*.

Connector *IF_MI*

Role *IFactory* = $\overline{\text{start}}$ →
 $(\overline{\text{calibrate}} \rightarrow \overline{\text{IFactory_normal}} \rightarrow \overline{\text{IFactory}})$

Role *Mitem* = $\overline{\text{calibrate}}$ → *Mitem*
 $_start \rightarrow (\overline{\text{normal}} \rightarrow \overline{\text{Mitem_calibrate}} \rightarrow \overline{\text{Mitem}})$

Glue = $\overline{\text{IFactory.start}} \rightarrow \overline{\text{Mitem.start}} \rightarrow$
 $(\overline{\text{IFactory.normal}} \rightarrow \overline{\text{Mitem.normal}} \rightarrow \overline{\text{Glue}})$
 $_ \overline{\text{IFactory.calibrate}} \rightarrow \overline{\text{Mitem.calibrate}} \rightarrow \overline{\text{Glue}}$

Connector *MI_MV* (*numMVs*: 1..)

Role *MI* = $\overline{\text{performCalculation}}$ → *MI* $_ \overline{\text{calibrate}}$ → *MI*

Role *MValues*_{1..numMVs} = $\overline{\text{perform_calculation}}$ → *MValues*
 $_ \overline{\text{calibrate}}$ → *MValues*

Glue = $\overline{\text{MI.perform_calculation}}$ →
 $\forall i : 1..numMVs ; \overline{\text{MValues}_i.\text{performCalculation}} \rightarrow \overline{\text{Glue}}$
 $_ \overline{\text{MI.calibrate}} \rightarrow \forall i : 1..numMVs ; \overline{\text{MValues}_i.\text{calibrate}} \rightarrow \overline{\text{Glue}}$
 $_ \S$

Connector *MI_Act* (*numAct*: 1..)

Role *MI* = $\overline{\text{actuate}}$ → *MI*

Role *Actuators*_{1..numAct} = $\overline{\text{actuate}}$ → *Actuators*

Glue = $\overline{\text{MI.actuate}}$ → $\forall i : 1..numAct ; \overline{\text{Actuators}_i.\text{actuate}} \rightarrow \overline{\text{Glue}} _ \S$

Step 6: Identify the variations of the one component (each subclass of a root class) and what other components related to the component must be changed. In the first step we identify abstract classes in the case of having class hierarchies to have the components in the first description. But this component is just one prototype of other components that can be mapped from the subclasses. Thus, if we have a class hierarchy, the idea is just to take each subclass, to see what other classes are related and then map them as components. Then, repeat the process from the step 3 until getting different versions of the description. In our example, the studied framework is small and all the classes were mapped in the first run of this algorithm. But in frameworks in large-scale, this step can be applied.

Step 7: Identify the components that represent hotspots and frozen spots. This step is focused to identify which components and connectors are fixed (this situation can be detected in the different descriptions obtained from the Step 6) and which ones are candidates to be changed in terms of a framework instantiation. In the microarchitecture that we are studying, all the components are frozen spots except the actuator which can customize its behaviour in the process *actuate*.

Step 8: At this step, we have a first level of description. We can identify predefined architectural styles in terms of set of classes or just components and connectors with an specific behaviour. For example it is possible to prove that the relationship between Measurement Item and Measurement Values is managed by the connector, when the measurement item needs information from their values, just only send the event to the connector and the connector makes the task of forwarding the same request to all the measurement values and also process the final results for the Measurement Item obtained from all the partial results of the Measurement Values

Step 9: We run the tool to verify the different properties in Wright.

1. Port Computation Consistency (component)
2. Connector Deadlock-Free (connector)
3. Roles Deadlock-Free (role)
4. Single Initiator (connector)
5. Initiator Commits (any process)
6. Parameter Substitution (instance)
7. Range Check (instance)
8. Port-Role Compatibility (attachment)
9. Style Constraints (configuration)
10. Style Consistency (style)
11. Attachment Completeness (configuration)

This step just only ensures us that our description is valid using Wright ADL. If there is an error reported by the tool, we should check it following the format defined in the Steps described previously.

Step 10: Refine each component considering

- if we have a hierarchical composition of objects that work together (definition of microarchitectures). This step is concerned to discover if there is a component that is composed of other objects and the different services that it offers are made using these objects. All the objects must be inside the 'boundaries' of the main object to consider it as a hierarchical composition. This means that for example, we can consider in this Step the composition relationship between Measurement Values and Measurement Item because the item calls the services of the values and the values call the sensors to do it, and all the information is returned to the measurement item. The possibility of using hierarchical composition will let us this kind of description.
- If there is a set of events joined by a non-deterministic choice which indicates a decision of the component regarding an internal state (internal state of the component). This step is concerned with expressing all the information related to the component avoiding to have non-deterministic choices. For example, in the case of Measurement Item there is a decision of starting in normal or calibration state. In fact, this choice depends on a boolean variable of the Measurement Item. The possibility of using Abstract Machines B let us this modification in the component.

Step 11: New components (not necessarily mapped from domain concepts) can be discovered. This step is concerned with having a new level of description. If this situation happens, it is suggested to start to study the component as a microarchitecture and follow again the steps only with the new components.

If we are interested in refining the behaviour of the component, this step is also concerned with having detailed information about the defined behaviour protocol in the components. For example, in the method start of the Measurement Item there are calls to Calculation Strategy and Actuation Strategy because the Calculation strategy is used to coordinate what actions must be made for collecting and converting data from one representation into another and Actuation Strategy is used to coordinate what actions must be made for generating the necessary effects on the actuators.

Step 12: Definition of the interaction protocols in interface types and association of frozen spots and hotspots in styles. This step is concerned with identifying the set of events that belong to an interaction protocol and defining styles for the fixed part and variable points of the framework. This will let us to have a clear view of how the framework is composed and measures the impacts of possible changes in its structure and object behaviour.

5.4 Architectural Description – Second Case: Architectural Patterns + Wright Description

According to [LC98] the architecting phase comes very early in the development process of an application. As it was said previously, its purpose is to define the gross organization of the application in order to provide first solutions partially meeting the application requirements and reaching some non-functional qualities like reusability, adaptability or portability. If not prepared at the architectural level, most requirement and non-functional qualities cannot be met in the code level. This is the reason that the architectural decisions are most of the time hard to take and require deep expertise both in software engineering and in the domain under consideration.

The objective of the first algorithm is to provide a way to recover the qualities mentioned previously. But in fact, it results interesting to have also a mechanism that allows us to notice the architectural concepts during the process of building the framework. [LC98] calls this process as architecting phase and says that it involves the following tasks:

1. Performing a first domain analysis and understanding the requirements of the application under construction.
2. Designing an architecture providing first solutions meeting the application requirements and reaching targeted qualities.
3. Allocating requirements to components and connections
4. Representing the architecture
5. Analyzing and evaluating the architecture with regards to the requirements
6. Documenting and communicating the architecture

This phase is followed by a phase of implementation of the architecture.

The second algorithm presented in this section proposes to carry out this architecting phase together with the process of design and implementation of an object oriented framework component. The objective is to show all the variations of services provided by the component in each run of the algorithm. As in the first algorithm, we use an example to make the algorithm understandable.

5.4.1 Example: A Generic Coordination Abstraction for Managing Shared Resources [CTN97]

A Coordination Component Framework for Open Distributed Systems was proposed in [CTN97]. They introduce different solutions to coordination problems, and each solution is embodied in a component in the framework. One of these components is used to provide solution to the management of shared resource.

In [CDK94], they define that there are some common elements which define the structure of these solutions : the definition of the resource, and the definition of the allocation policy. Some features of these elements are fundamental for the specification of the solutions. For the resource they are basically : the size of the resource, and the number of concurrent entities which can access the resource at the same time. For the allocation policy they are basically : the order in which the resource is going to be assigned, the maximum allowed to each entity, if there will be priorities on the requests of the allocations, and what to do with the allocation requests that cannot be processed simultaneously (to define a waiting queue of entities, to ignore them, etc.). There are also some fault-tolerance aspects like what to do in case of software failures (to ignore precedent request, to guarantee recovery, etc.) that have taken into account. All these aspects define the parameters of variability of the generic coordination abstraction that could be specified and that can be used to generate specific coordination solutions.

One example shown to see how the component is applied is a toy banking system. In this case the resource is an account database which is shared by multiple teller machines. The teller machines need to get information from the account database in order to check a client's account.

They also need to update account information if they have given money to a client. To keep the database consistent we need an access policy to regulate the multiple requests.

As a solution for this regulation they introduce an access policy component. This solution not only provides access regulation, but also explicitness of architecture and flexibility.

5.4.2 Requirements

The main goal of the design is to implement the access policy to the database in a reusable and configurable way. In terms of the component oriented approach, this means that we want to have a structure where we can plug in different policies, without having to change other parts of the solution. Three kinds of policies are distinguished:

- *Policies that need no other information about the requests:* This is the easiest kind of policy. They can do their job without any knowledge about the commands they have to dispatch. A typical example is a FIFO policy: no matter what commands come in, the policy just dispatches them in order they come in.
- *Policies that need type information about the requests:* This kind of policy depends on the “nature” of the command: every *commandType* has one or more properties which are needed by the policy. A typical example for this case is the readers/writers policy. The policy has to know if a command is a reader or a writer command. So this information must be made available in our solution.
- *Policies that need external information about the requests:* This kind of policy depends on information which can be different for each instance of a command. It can be thought as a priority policy, where, depending on the sender of the request, a command has a certain priority. Again, somehow this information must be available to the policy.

5.4.3 Solution

There is an Interface of the solution to the rest of the application. Clients have to call this *Interface* to access the access solution. The resource itself is also modelled and there is a part which represents the *control policy*. To give the policy the ability to buffer the commands, change their order or execute them in parallel, it is necessary an explicit representation of the commands. This is done using the Command Pattern [GHVJ95].

Then let's see the relationship between *Command* and *Policy*. For the first kind of policy, the basic Command pattern suffices. For the second kind of policy, we need to make type information available at run-time in order to be able to use this information at run-time, for instance to link this information to certain policy-dependent properties. We make this information available by providing a *CommandType* class to every subclass of the abstract *Command* class.

For the third kind of policy we need to make information available which can differ for every instance of a command. We do this by connecting a *Property* class to every *Command* class and the addition of a *SetProperty* and a *GetProperty* method to the *Command* class. What the exact information is, that subclasses of this class represent, is totally dependent on the application in which it is used. It could be that the name of the sender of the request is made available. The information, that is made available, should not be information especially linked to a policy. As an example, we take again the priority policy. In the *Property* object, we should make the sender of a request available. The linking of this sender with a priority is done later at the policy. We do this to keep the *Command* as independent of the policy as possible: the information is really a property of the *Command* and this information can be used by different policies.

The *Policy* part is set up using the Strategy pattern [GHVJ95]. The hard part is again the fact that we have to deal with the application dependent information. The solution is to represent this information explicitly in so-called “*configuration*” objects. These configuration objects contain at run-time the information that is needed by the policies. So we may have, for example, a configuration object that contains of a link between command types and a property *isReader*. Another example is an object that links the names of possible request senders to a certain priority. We see here the difference between information that is made available in a *Property* object and

that in a Configuration object. The former is general information, the latter contains the policy dependent information.

In figure 5.7, we see the total design for the shared resource access policy solution. The class Interface is the interface to the resource for the rest of the system. For every command which is invoked by an incoming events, a Command object is created. These commands are then given to the policy which is connected (through parameterization) to the interface. This policy handles the commands, i.e. determines when and in which order the request can access the resource. If the command is allowed, it is executed.

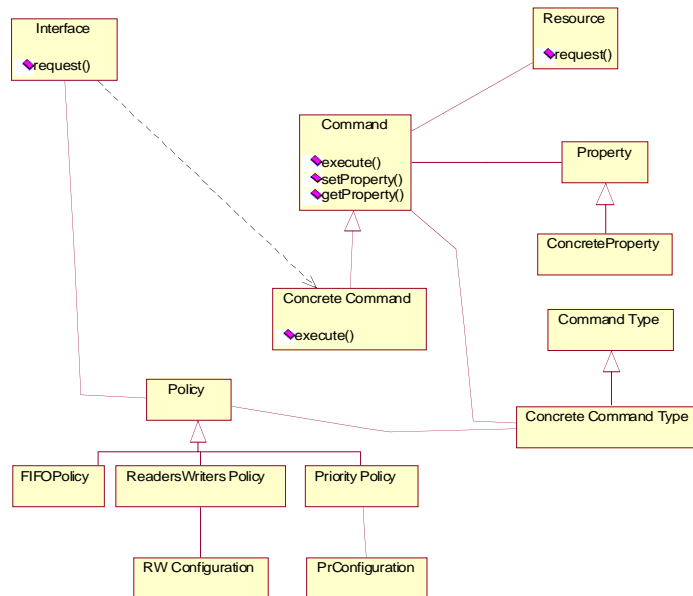


Figure 5.7: Class Model for the Shared Resource Access Policy

5.4.4 Approach: Use of Steps

Step 1: Can we apply any known architectural pattern in our model ?

It is possible to use the guidelines proposed in [BMR+96] and [SG96] to identify some architectural patterns that can be applied:

- Control decomposition of an overall system task into cooperating subtasks: Layers, Pipers and Filters, Blackboard
- Infrastructure for distributed applications: Pipes and Filters, Microkernel, Broker
- Support the structure of software systems that feature human-computer interaction: Model-View-Controller, Presentation-Abstraction-Controller
- Support the extension of applications and their adaptation to evolving technology and changing functional requirements: Microkernel, Reflection.

Clearly, as we explained in the chapter 2, the domains and the models allow us to apply none, one or a combination of several architectural patterns. Unfortunately, we can not give an example of the use of an architectural pattern with the framework that we use as example in this algorithm. But we studied other components' frameworks in a larger scale, and we discovered a large applicability of the Layers Architectural Pattern. For example, in [BGK+97] they document the development of a framework for application covering almost the entire area of banking: tellers, loans, stocks and investments departments as well as self-services facilities. In the class model, they use layered framework (shown in the figure 5.8), where the Application Layers provide the

software support for the different workplace contexts, Business Section Layers consists of frameworks with specific classes for each business section and Business Domain Layer contains the core concepts for the business as a whole.

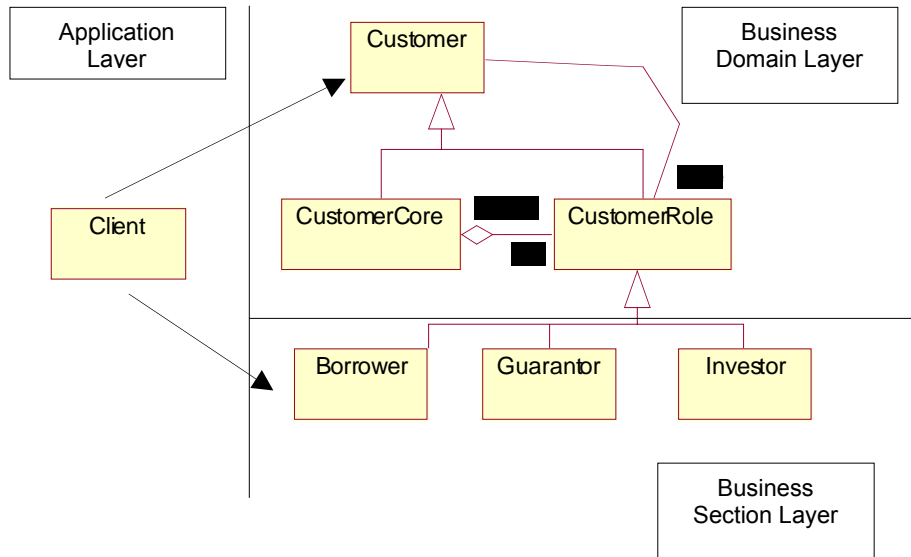


Figure 5.8: Example of Application of Layers Architectural Pattern

Next steps of the algorithm are focused on the description of each layer or each component in the framework.

Step 2: Classify and categorize your abstract classes in the model of connectors and components.

In this step, the objective is to define the semantic function of the different classes inside the model. We define *semantic function* as the kind of task that the class fulfils inside the model, avoiding to have just only a set of classes without any meaning for the developer or the designers. The main idea is to divide the classes in two sets: units of computation (components) and units of communication (connectors). To do this process, we propose:

1. To make a preliminar mapping of each class in a component according to their activity in the model: processing, data, connecting, controller, coordinator, interface, service provider, information holder or structuring components. In our example, we classify Command and Resource classes as data components, Policy as coordinator component and Interface as connecting component (although it can be classified as coordinator or interface component).
2. To "reclassify" some components to convert them in connectors following the table 1. This conversion is a variant using a mixture of the categorization of components presented in the chapter 4. In our case, the result is Command and Resource are components and Interface and Policy are connectors.

<i>Preliminar Mapping</i>	<i>Final Mapping: Architectural Element</i>
Processing Components	Component
Data Components	Component
Connecting Components	Connector
Controller Components	Connector
Coordinator Components	Connector
Interface Components	Connector
Service Provider Components	Component/Connector
Information Holder Components	Component

Structuring Components	Component
------------------------	-----------

Table 5-1: Mapping for Architectural Element

Step 3: Study the relationships between the different classes represented as components to map them as connectors. The last step allows to identify classes as components or connectors, but in fact, there are classes mapped as components and they do not have any evident connectors, or any class which fulfills this function. The idea of this step is to capture these relationships in a connector. The reason is that we must express the interrelationships between components as explicit connectors with Wright ADL. As it was defined in the section about Mappings for Connectors, there are three ways in which classes A and B: instances of B can be instances/class variables of A, instances of B are used as parameters in one method of A and instances of B are connected to an instance of A by a dependency mechanism. In these cases the connectors can be as simple as sender-receivers of events without any other events, in the best case the connector will have a function such as distribution of events. In our example, the Command component has the former relationship with Resource, so we express this relationship as a connector.

Step 4: Using Wright ADL, define the protocol of the components or connectors based on the information of the classes and their possible interactions.

This step help us to identify the set of messages and different states of the object that we need in the model. From our example, we take two connectors (Interface and Policy) and two components (Command and Resource) to analyze.

Connector Interface: This element represents the entry point to the framework component, so we need two events: one to indicate that we received a request to be executed (*request*) and one to indicate that this request has to be processed according to the policy (*create*). The event *request* is an observing event and the event *create* is an initiating event. This connector represents the relationship between the external users of the component and the Command component.

Component Command: This element represents the request that will be executed in the resource. It has three events: *create* (indicates that it was created), *put* (indicated that it is enqueued in the policy to wait for the moment to be executed) and *execute* (indicates that it can be executed). The events *create* and *execute* are observing events and the event *put* is an initiating event.

Component Resource: This element in any context can answer to three states associated to events: *free* (it can accept any request), *request* (it receives the request), *busy* (it is processing a request).

Connector Policy: This element represents the algorithm that administrates the order of execution of the commands. This connector receives the Commands components and gives the order of execution to work on the resource. It has associated three events: *put* (indicates that it received a Command), *execute* (indicates that a Command can start to work on the resource), and *ready* (indicates that the resource is free).

Step 5: Model the set of events in the Ports of the components and the Roles of the conectors and the chain of events in the Computation and Glue of the components and connectors respectively. Following from the last step we show the definition of the protocols of the two components and the two connectors.

Connector Interface: We assume that we have only one type of request to be executed.

Connector Interface

$$\begin{aligned}
 \mathbf{Role\ User} &= \overline{\text{request}} _ _ \$ \\
 \mathbf{Role\ Command} &= \text{create_Command} \\
 \mathbf{Computation} &= \text{User.request_} \overline{\text{Command.create_}} \mathbf{Computation}
 \end{aligned}$$

Component Command: The restriction is that once it is created, it must be enqueued in the policy queue and has to wait for its turn to execute. This restriction is shown in the Computation

$$\begin{aligned}
 \mathbf{Component\ Command} \\
 \mathbf{Port\ In} &= \text{create_In} \\
 \mathbf{Port\ Out_Policy} &= \overline{\text{put}} _ \text{Out_Policy_execute} _ \overline{\text{ready}} _ \$ \\
 \mathbf{Computation} &= \text{In.create_} \overline{\text{Out_Policy.put}} _ \text{Out_Policy.execute} \\
 &\quad _ \text{"execute the work on the resource"} _ \overline{\text{Out_Policy.ready}} _ \$
 \end{aligned}$$

Connector Policy:

$$\begin{aligned}
 \mathbf{Connector\ Policy} \\
 \mathbf{Role\ Command} &= \overline{\text{put}} _ \text{Command_execute} _ \overline{\text{ready}} _ \$ \\
 \mathbf{Computation} &= \text{Command.put_} \overline{\text{Command.execute}} \\
 &\quad _ \text{Command.ready_} \mathbf{Computation} _ \$
 \end{aligned}$$

Component Resource: The only condition to be fulfilled is that while the resource is available sends the event *free*. But when it receives the request, it has to notify that the state is busy and that can not execute the request event. Both conditions are shown in the computation of the component.

$$\begin{aligned}
 \mathbf{Component\ Resource} \\
 \mathbf{Port\ In} &= \text{execute_In_} \overline{\text{free}} _ \text{In_} \overline{\text{busy}} _ \text{In_} \$ \\
 \mathbf{Computation} &= \overline{\text{In.free}} _ \text{In.execute_} \mathbf{Continue} _ \mathbf{Computation} \\
 \text{where } \mathbf{Continue} &= \text{In.execute_} \overline{\text{In.busy}} _ \mathbf{Continue} _ \$
 \end{aligned}$$

Step 6: Use the Wright tool to check all the properties in a description and solve the possible problems.

1. Port Computation Consistency (component)
2. Connector Deadlock-Free (connector)
3. Roles Deadlock-Free (role)
4. Single Initiator (connector)
5. Initiator Commits (any process)
6. Parameter Substitution (instance)
7. Range Check (instance)
8. Port-Role Compatibility (attachment)
9. Style Constraints (configuration)
10. Style Consistency (style)
11. Attachment Completeness (configuration)

This step just only ensures that our description is valid using Wright ADL. If there is an error reported by the tool, we should check it following the format defined in the steps defined previously.

Step 7: Analyze the interface where there is a non-deterministic choice to see if the component must have an internal state to take some decisions.

This step can be optional because we can change the non-deterministic choice in a deterministic one using an Abstract Machine B, getting a new level of the description. Or we do nothing and we solve this problem in the implementation itself. For example, in the Resource component we can specify the state of working or not.

Component *CH_Resource*

Port *In* = $\overline{\text{execute}} _ \text{In} _ \overline{\text{free}} _ \text{In} _ \overline{\text{busy}} _ \text{In} _ \$$

Computation

Component *Resource*

Port *In* = $\overline{\text{execute}} _ \text{In} _ \overline{\text{free}} _ \text{In} _ \overline{\text{busy}} _ \text{In} _ \$$

Port *Component* = $\overline{\text{is_working}} _$
 $(\overline{\text{working}} _ \text{Component} _ \overline{\text{not_working}} _ \text{Component})$
 $_ \overline{\text{set_value?value}} _ \text{Component}$

Computation = $\overline{\text{In.free}} _ \text{In.execute} _ \overline{\text{Component.set_value!True}}$
 $_ \overline{\text{Continue}} _ \text{Computation}$

where **Continue** = $\overline{\text{In.execute}} _ \overline{\text{Component.isWorking}}$
 $_ (\overline{\text{Component.working}} _ \overline{\text{In.busy}} _ \text{Continue})$
 $_ \overline{\text{Component.not_working}} _ \overline{\text{In.free}}$
 $_ \overline{\text{Component.set_value!False}} _ \$$

Component *AM_Resource*

Port *B* = $\overline{\text{is_working}} _ \text{B} _ \overline{\text{set_value?val}} _ \text{B}$
 $_ \overline{\text{working}} _ \text{B} _ \overline{\text{not_working}} _ \text{B}$

Connector *B_Resource*

Role *B* = $\overline{\text{is_working}} _ \text{B} _ \overline{\text{set_value?val}} _ \text{B}$
 $_ \overline{\text{working}} _ \text{B} _ \overline{\text{not_working}} _ \text{B}$

Role *Component* = $\overline{\text{is_working}} _$
 $(\overline{\text{working}} _ \text{Component} _ \overline{\text{not_working}} _ \text{Component})$
 $_ \overline{\text{set_value?value}} _ \text{Component}$

Glue = $\overline{\text{Component.set_value?True}} _ \overline{\text{B.set_value!True}} _ \text{Glue}$
 $_ \overline{\text{Component.set_value?False}} _ \overline{\text{B.set_value!False}} _ \text{Glue}$
 $_ \overline{\text{Component.is_working}} _$
 $(\overline{\text{Component.working}} _ \text{Glue}$
 $_ \overline{\text{Component.not_working}} _ \text{Glue})$

Instances

State = *AM_Resource*
I_Resource = *Resource*

Attachments

State.B as *B_Resource.B*
I_Resource.Component as *B_Resource.Component*

End Resource

Bindings

I_Resource.In = In
End Bindings

Step 8: Translate the description in an implementation following the mapping from CSP to Smalltalk/Java code presented previously.

Step 9: Analyze the points of variability in the set of components and connectors resulting from the first description. Build the new variations in terms of the components and connectors with the class model in parallel.

This step consists of identifying which changes are necessary to get a specialized behaviour of the component. In the example, we detected three types of changes:

- Extension of the protocol of a component or a connector.
- Use the same protocol but changing the behaviour of the component/connector.
- Objects that are not modified and are useful in all the proposed descriptions.

In the first case, we talk about hierarchical classes and in the second case we can infer that it is possible to use a Strategy pattern, where all the algorithms are encapsulated in classes, or a Composite Pattern, where the behaviour of some messages are different if we have the Composite or the Atomic object. These two variations show the hotspots of the component and the last one the frozen spots of the component that we map in the description.

For example, in our example the first variant is to have a set of operations, e.g. read and write. The fact that we have two operations with conflicts between them (an operation of writing can not be made during reading process, and reading process can not be taken into account when a writing process is happening) changes the definition of the protocols of Resource, Interface and Policy. In all the cases, we must identify what kind of operations is to know how to proceed.

Connector Interface

Role *User* = $\overline{\text{read}} _ \text{User} _ \overline{\text{write}} _ \text{User} _ \$$
Role *CommandReader* = $\text{create} _ \text{CommandReader}$
Role *CommandWriter* = $\text{create} _ \text{CommandWriter}$
Computation = $\overline{\text{User.read}} _ \overline{\text{CommandReader.create}} _ \text{Computation}$
 $_ _ \text{User.write} _ _ \text{CommandWriter.create} _ _ \text{Computation}$

Connector Policy (nbr: 1., nbw: 1.)

Role *CommandReader*_{1..nbr} = $\overline{\text{put}} _ \text{CommandReader}$
 $_ _ \text{execute} _ _ \overline{\text{ready}} _ _ \text{CommandReader} _ _ \$$
Role *CommandWriter*_{1..nbw} = $\overline{\text{put}} _ \text{CommandWriter}$
 $_ _ \text{execute} _ _ \overline{\text{ready}} _ _ \text{CommandWriter} _ _ \$$
Computation = $(\exists i..k < \text{nbr} \parallel \overline{\text{CommandReader.put}} _ _ \overline{\text{CommandReader.execute}})$
 $_ _ \forall i..k _ _ \overline{\text{CommandReader.ready}} _ _ \text{Computation}$
 $_ _ \overline{\text{CommandWriter.put}} _ _ \overline{\text{CommandWriter.execute}}$
 $_ _ \overline{\text{CommandWriter.ready}} _ _ \text{Computation}$
 $_ _ \$$

The Component Resource must not pay attention to inform that the operation has finished, because in this case it can accept a set of operations of the same type (e.g. readers at the same time, or just only one writer).

Component Resource (nbr: 1., nbw: 1.)

Port *In* = $\text{execute} _ _ \text{In} _ _ \$$
Computation = $\overline{E} \ i:1..k \parallel \text{In.execute} _ _ \text{Computation}$

Step 10: Repeat the steps 8-9-10 to get different variations of the component and their implementations.

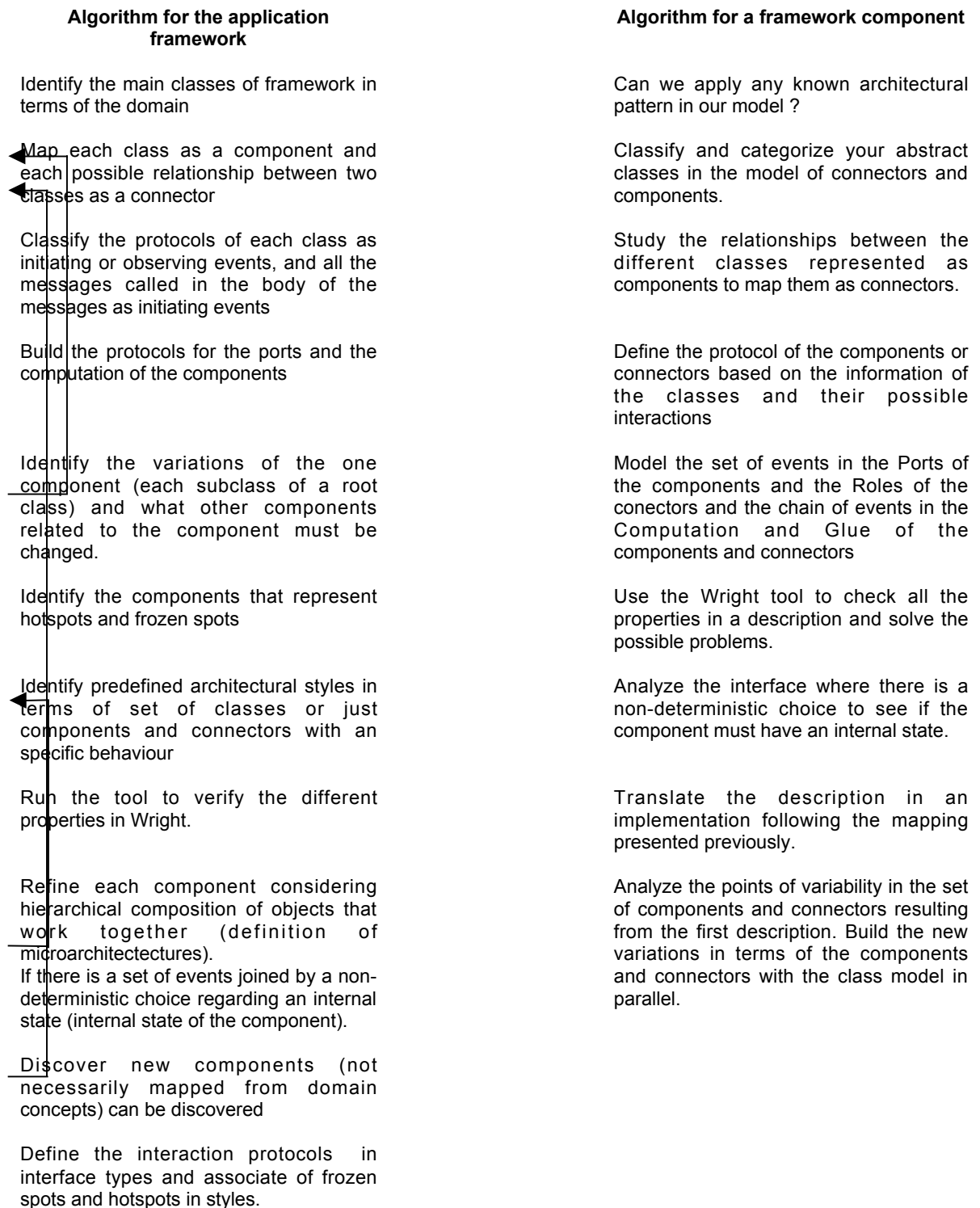


Figure 5.9: The developed algorithms seen graphically

5.5 Summary

In this chapter, we presented the definition of the mapping from Java/Smalltalk code to CSP-notation and two algorithms to get an architectural description for an object oriented application framework and a framework component. We also describe the applicability of these algorithms with two specific examples to make them more understandable. This work lets us to see specifically where we found problems to use architectural patterns and Wright ADL to inference a semantic structure of an object oriented framework (component).

Chapter 6: Architectural Description Aspects: Discussion

The formal approaches such as Wright ADL and architectural patterns can be considered as complementary techniques to get an architectural description of an object oriented framework. The two algorithms presented in the previous chapter give the proof of this fact. But we have found some interesting aspects (during the development of the algorithms) that should be analyzed separately. This chapter proposes a discussion of them with the goal of giving a complete panorama and conditions before the analysis of evolution impacts in our approach.

6.1 Analogy between Architectural Style and Framework Architecture

In the first algorithm the last step proposes: "Define the interaction protocols in interfaces types and associate of frozen spots and hotspots in styles". Let's see the reasons and the consequences of this decision. To do this, then let's consider the following definitions,

- ⇒ An *application framework* is a set of classes the embodies an abstract design for solutions to a family of related problems [Joh92].
- ⇒ An *architectural style* defines a family of systems that have a common architectural design vocabulary and satisfy a common set of design constraints [All97].

Defining a software architecture for a framework (component), we identify the frozen spots and the hotspots. They represent the fixed part and the points of variability of the frameworks. In the case of application frameworks, if we model the classes as components and the relationships between them as connectors, in fact, the fixed part can be considered as the architectural style of the framework because these components express the common design vocabulary and set of design constraints for all the applications resulted from the framework instantiation. For example, in the case of the Measurements System Framework, all the classes are frozen spots except the behaviour of the strategies (CalculationStrategy, UpdateStrategy and ActuationStrategy classes). So, we can define an architectural style considering:

- Components and connectors defined as components and connectors types
- The set of events of each port and role in the components and in the connectors respectively defined as interface types
- The constraints of how the components and the connectors must be related.
- A cardinality indicating how many components (mapped from the classes) we need in a possible instantiation of the framework, represented by a definition of instances and attachments in a Wright configuration.

Let's see one example where we present the architectural style partially:

{ Specification of Interface Types for the Components' Ports }

Interface Type *TInFactory* = *trigger* → *InFactory* *calibrate* → §

Interface Type *TOutFactory* = *start* → (*calibrate* → *OutFactory* _ *normal* → *OutFactory*)

Interface Type *TInItem* = *start* → (*normal* → *TInItem* _ *calibrate* → *TInItem*)

Interface Type $TOutItem = \overline{\text{performCalculation}} \rightarrow TOutItem \quad \overline{\text{calibrate}} \rightarrow TOutItem$
 $\quad \overline{\text{actuate}} \rightarrow TOutItem$

Interface Type $TIFactory = \overline{\text{start}} \rightarrow (\overline{\text{calibrate}} \rightarrow IFactory \quad \overline{\text{normal}} \rightarrow IFactory)$
Interface Type $TMitem = \overline{\text{calibrate}} \rightarrow Mitem \quad \overline{\text{start}} \rightarrow (normal \rightarrow Mitem \quad \overline{\text{calibrate}} \rightarrow Mitem)$

{ Specification of Components }

Component $Item_Factory$

Port $In = TInFactory$

Port $Out = TOutFactory$

Computation $= In.trigger \rightarrow \overline{Out.start} \rightarrow (\overline{Out.calibrate} \rightarrow Computation$
 $\quad \overline{Out.normal} \rightarrow Computation)$
 $\quad \overline{In.calibrate} \rightarrow Computation$
 $\quad _ \$$

Component $Measurement_Item$

Port $In = TInItem$

Port $Out = TOutItem$

Computation $= In.start \rightarrow (In.calibrate \rightarrow \overline{Out.performCalculation} \rightarrow \overline{Out.calibrate}$
 $\quad In.normal \rightarrow \overline{Out.performCalculation} \rightarrow \overline{Out.actuate})$
 $\quad \rightarrow Computation$
 $\quad _ \$$

{ Specification of Connectors }

Connector IF_MI

Role $IFactory = TIFactory$

Role $Mitem = TMitem$

Glue $= IFactory.start \rightarrow \overline{Mitem.start} \rightarrow (IFactory.normal \rightarrow \overline{Mitem.normal} \rightarrow Glue$
 $\quad _ IFactory.calibrate \rightarrow \overline{Mitem.calibrate} \rightarrow Glue)$

{ Specification of the Constraints }

Constraints

$\exists \text{ factory} : \text{Components_Type}(\text{factory}) = IFactory \wedge \#\text{factory} = 1$
 $\quad \wedge p : \text{Ports} \mid p _ \text{Ports}(\text{factory})$
 $\quad _ \text{Type}(p) = TInFactory \vee \text{Type}(p) = TOutFactory$

$\exists \text{ item} : \text{Components_Type}(\text{item}) = Measurement_Item \wedge \#\text{item} = 1$
 $\quad \wedge p : \text{Ports} \mid p _ \text{Ports}(\text{item})$
 $\quad _ \text{Type}(p) = TInItem \vee \text{Type}(p) = TOutItem$

$\exists \text{ factory_item} : \text{Connectors_Type}(\text{factory_item}) = IF_MI \wedge \#\text{factory_item} = 1$
 $\quad \wedge r : \text{Roles} \mid r _ \text{Roles}(\text{factory_item})$
 $\quad _ \text{Type}(r) = TIFactory \vee \text{Type}(r) = TMitem$

$\forall r : \text{Roles} \mid r _ \text{Roles}(\text{factory_item})$

$$\begin{aligned} & \exists p1 : Ports \mid p1_Ports (Item_Factory) \\ & \quad ((Item_Factory, p1), (factory_item, r)) \quad \mid \quad Attachments \\ \wedge \exists p2 : Ports \mid p2_Ports (Item_Factory) \\ & \quad ((Item_Factory, p2), (factory_item, r)) \quad \mid \quad Attachments \end{aligned}$$

In the case of a framework component, as it is thought as a black-box entity, the architectural style will be used in the configuration of the system where the component will be used.

This approach of separating fixed from variable parts in an architecture is similar to a proposal made in [Mac00] and in [Neb98].

[Mac00] works with product families and they propose the existence of a reference architecture and a configuration architecture in product families architecture. The reference architecture describes the style upon which every product is built. It lists the rules for interaction of the various components, and the abstract framework that holds them together. A reference architecture may contain information, constraints and guidelines on the infrastructure that needs to be present in order to hold together the various components. This could mean the architectural structure, or the glue that holds the components together into the same system. Meanwhile, the configuration architecture is described for every product by the product configuration architecture document. In the case of several products that belong to the same family, a family configuration architecture document naturally springs from the combination of all member product configuration architecture documents.

An approach closer to the developed one in this dissertation is [Neb97] which proposes two complementary notions of software architecture. They represent a separation of concerns relating to the distinct nature of the information encompassed by each kind of architecture. The initial distinction is between classes and instances. A domain architecture consists of a set of classes (components) that are connected by potential relationships (connectors) between classes as expressed by its set of dependencies. The term domain reflects the fact that this notion of architecture tells you how the model of your problem domain is structured. But a situation Architecture consists of a set of instances (components) that are connected by the actual relationships (connectors) between these instances. The term situation architecture reflects the fact that a situation architecture represents the structure of a particular concrete situation from the problem domain as defined by a configuration of instance and their actual relationships. This is in contrast to a domain architecture, which represents potential relationships. The relationship between a domain and situation architecture is one of instantiation. A situation architecture is an instance of a domain architecture. The situation architecture's components and connectors are instances of the domain architecture's components and connectors. Two kinds of architecture are complementary but they are fundamentally different. One way of thinking of the relationship is that the domain model defines what can and can not happen while a situation model defines, in conjunction with any input, what does and does not happen.

In our approach, the reference architecture [Mac00] and domain architecture [Neb98] could be compared to the definition of our architectural style of the framework, where we express which the classes (frozen spots: different aspects already implemented) are in all the applications resulting from the framework instantiation. In fact, in [Neb98], the mapping for relationships between classes are closer to our working level (code-level) and also a lower level of description

In our approach, we do not have a configuration architecture [Mac00] and situation architecture [Neb98]. In fact, we can think that this is similar to the configuration description made with Wright ADL to represent an instantiation of a framework. In this case, we use the defined architectural style and we define the components and connectors that are specific to the instantiation.

6.2 Levels of Description

In both algorithms we propose an incremental process of the growth of the architectural description starting from the first architecture obtained. There are two possible issues of growth:

- refinement (or grouping) of components getting microarchitectures and
- replacement of a component (mapped from a class) by another one that represents a refinement in the protocol of the component (a subclass).

We consider that in fact, we have two dimensions of growth and each dimension generates levels of description. If we consider both dimensions in X-Y axis, we can have a parameter to measure qualities of our design and/or implementation mapped to an architectural description. Supposing that we build a X-Y cartesian axis where (Figure 6.1):

- Y-axis gives information about how many architectural descriptions we have obtained refining (or grouping) components and connectors (r_1, r_2, \dots, r_n)
- X-axis gives information about how many changes in components and connectors (mapped from classes) we have made in our base architectural description (s_1, s_2, \dots, s_m)

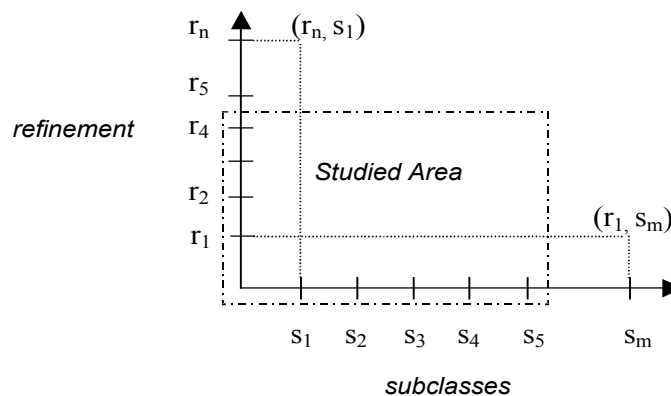


Figure 6.1: Growth of the Architectural Description in X-Y axis

Let's see then what kinds of evaluation we can make about our architectural description.

An architectural description located in (s_m, r_1) indicates that we have a deep tree representing hierarchy of classes and the absence of microarchitectures (grouping of classes related to fulfill a task). This can be a signal of 'bad design' caused by too many classes (related by a hierarchy) and few responsibilities in each subclass.

An architectural description located in (s_1, r_n) indicates that we have a set of classes (without siblings in a hierarchy), each of them is an entry point for a microarchitecture. Here we have different possibilities. If we are modelling a components' framework, each refinement represented in the Y-axis will indicate a refinement in one different component each time. This means that the framework is well-built because in those cases, the components are built as black-box entities and they are like microarchitectures that are adapted by the interface offered by the component. But if we are modeling an application framework, the point (s_1, r_n) in the diagram shows that we have a set of classes (very few because we are in the first step of growing in the X-axis), and the classes have too many services each of them and several classes are dependent of the main classes (big microarchitectures). This can be a signal that the classes should be divided with different responsibilities in the classes which are the result.

We can also use the diagram to locate regions of what we consider a large or a small framework in terms of amount of classes and microarchitectures. We can also decide which of these regions are better to determine a good balance of classes as lonely units and entrypoints to microarchitectures. In our case, we have studied a region limited by (s_0, r_0) , (s_0, r_4) , (s_5, r_0) and

(s_5, r_4) where the application framework and the framework components have around 20-25 classes.

6.3 Level of Components

In the mapping from classes to components, we refine the relationships of the classes until the level of management of simple datatypes (integer, char, etc). We adopt this level of detail, because

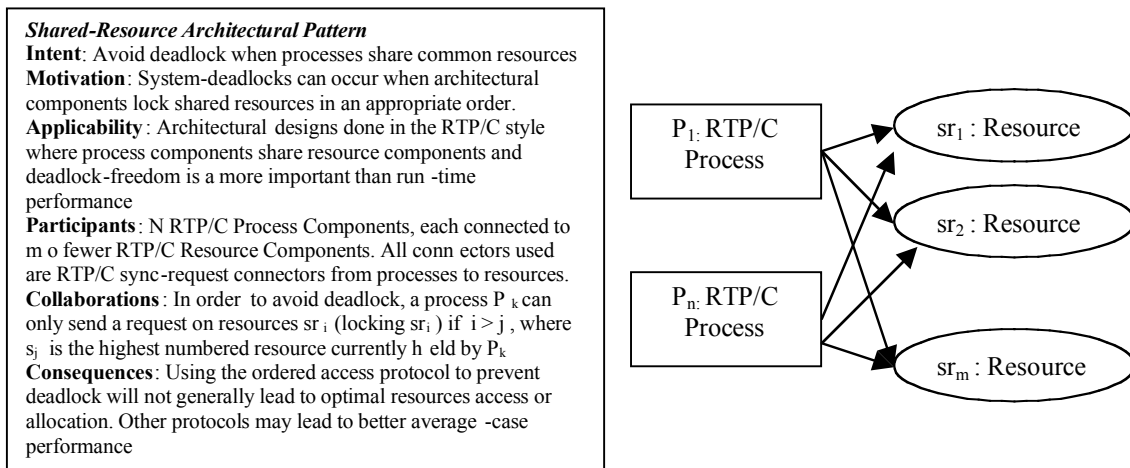
- We are working at the code-level
- We are limited by the non-inferenced information of the parameters provided by Wright ADL

But in fact, we are not constrained to work in this level of detail. Going back to the example of the class model of Truck presented in chapter 5, if we are sure that we will not work with the information about the Manufacturer, we can work with this component as a simple datatype parameter, such as for example: *Truck (driver, manufacturer, year)*

6.4 The Potential of the Connectors

Wright ADL expresses explicitly the connections between components through the use of connectors. The focus of this ADL is in the connectors. In the developed example, most of the connectors are only just receivers and senders of events. But in fact, they can be considered the 'core' of the application because they organize all the behaviour and interaction between all the classes (mapped as a components).

[MKMG96] proposes that the structure for pattern description provided in [GHJV95] for describing architectural patterns of the RTP/C style. Let's see in the figure one example of the proposal.



Clearly, the examples identify different ways of connection offered by the style. Thinking in connectors defined with Wright ADL, we can think in generating a catalog of them and promotes the reuse of them having into account that all formal properties have been proven in the defined structures. For example, we could have a catalog of connectors that models the different policies to access a shared resource.

6.5 Increment of Connectors and Components

In the first algorithm, we present two approaches: use of predefined styles and start from the “scratch” with the description. Comparing both approaches developed, we see clearly an increment of the amount of components and connectors. But, in fact, the main idea points out to represent close all the framework classes and their interaction. After the analysis of the microarchitectures and styles inside the description, we can reduce this amount getting a description according to our needs of description

6.6 Management of Errors

The static nature of the Wright ADL and the typed protocol makes us difficult adding management of errors. The problem is focused that we must modify the protocol of the ports and computation, and this can provoke a great impact in all the description. In the case of modification, we can do it trying to focus the changes in the microarchitectures and also running the tool for checking that our description is valid.

6.7 Lack of Expressiveness in Structural Features

The structural features can not be expressed. In both algorithms, we established that the description was composed of our classes mapped to the domain concepts. The hierarchy relationship between two components can not be expressed explicitly, this means that we can change a component by another one which adds behaviour to the first one (subclass relationship) but losing the idea of that one is subclass of the another one.

6.8 Lack of Expressiveness of Timing

The Wright ADL does not allow us express timing relationships. This means expressing a service that is regulated by a timer, and that it is offered in regular time intervals. For example, this restriction can be seen when we work with the Measurement Systems Framework and we represent PeriodicUpdate class (subclass of UpdateStrategy class) used by the Sensor class. An instance of this class has a cycle that indicates that after a set time interval, it must indicate to the sensor that it must update its information about the measurement item. We express this relationship is related as non-deterministic choice between the different behaviours of an instance of UpdateStrategy class.

6.9 Dynamic Binding and Creation/Destruction of Components

The static nature of Wright ADL does not allow us to represent a dynamic binding of an object in a body method and the creation and destruction of components in the description. The first case can be illustrated with the example of ActuationStrategy in the method trigger in the Item Factory class. In this method, an instance of a Measurement item is initialized and one of the instance variables is an instance of Actuation Strategy class. This variable indicates if the instantiated measurement item is in normal state (after having taken the measurements of the physical item, the actuators are called) or in calibration state (after having taken the measurements

of the physical item, the item is taken as the new prototype for the rest of the items). But this behaviour at run-time is seen when the actuation phase arrives, but in our description we had to express this difference when we modelled the initialization of the item.

Following with the same example, the factory initializes one item each time that a physical item enters in the physical system. Here, we should have a creation/destruction of instances, but in the description we fix the relationship between these two classes and we simulate this process with the event *component.create*. But the notation should provide a way to express that this is a dynamic relationship between the two components.

6.10 Detection of Errors with the Wright Checking Tool

The focus of our approach was the use of Wright checking tool as a secondary tool to our algorithms. The tool can assure us that the obtained description is valid. In case of an error detection, while arguably one might attribute the detected problem to our specification, and not to framework, it does point out a place where the complexity of the specification can lead to errors that might be hard to detect otherwise. Without a precise model and effective automated analysis tools to identify problem areas, errors could easily be introduced, undetected, into an implementation

6.11 Use and Discovering of Patterns in the Architectural Description

In both algorithms we propose the discovering of architectural and design patterns during the inference of the architectural description. We must note that we have different levels of abstraction with patterns (from higher to smaller scale):

- ⇒ Architectural Patterns associated (or not) to architectural styles: Patterns and architectural styles are complementary mechanisms for encapsulating design expertise. As we said previously an architectural style provides a collection of building block design elements, rules and constraints for composing the building blocks, and tools for analyzing and manipulating designs created in the style. Styles generally provide guidance and analysis for building a broad class of architectures in a specific domain whereas patterns focus on solving smaller, more specific problems within a given style (or perhaps multiple styles) [Neb98]
- ⇒ Design Patterns: They are conceptual on a smaller scale than an individual architecture and numerous architectural patterns may be found in a single architecture. A design pattern is similar to a domain model in the sense that they both provide general solutions to a class of a related problems; however, the major difference is that a pattern is generic and will occur across many otherwise unrelated domain architectures [MKMG96].

The studied frameworks and the inferred description allowed us discover only a few patterns. The detection of an architectural pattern in a framework is a difficult task because all the design/implementation decisions were already made and the idea of applying one was not considered in the first design stages. The most applied is the Layered Architectural Pattern. The parts are separated into layers by responsibility or service, the increased complexity associated with large applications become spread across those layers. By applying a divide-conquer approach to complexity, the total number of connections per part can be kept reasonable. Components, in turn, become more reusable and certainly more understandable.

Luckily we can detect an architectural pattern associated to an architectural style and different design patterns to be joined to the generated architectural style at framework-level. The interesting issue about the detection of design patterns is to see how they can be represented with

formal languages and thus, having another way of considering them, not just only informal solutions to particular problems.

6.12 Properties of Cohesion and Coupling in the Architectural Description

[San97] proposes to evaluate the quality of the structure of an architecture introducing the concept of cohesion and coupling. The cohesion of a component makes it possible to measure the quality of its structuring. A component should aim to fulfill only one logical function or logical entity. Thus, all the parts of a component must allow this goal. We thus calculate the degree of cohesion of a component compared to the parts of this component which contribute to its logical function (if every part do it, then the component will have a high degree of cohesion). In our approach, the component are small (mapped from classes) and are not composed of other ones, but we make the analysis of microarchitectures which comprises a set of classes that are gathered to fulfill a task of the framework.

[San97] thus defined seven levels of cohesion. We define them and mention if they can be analyzed in our context

- Cohesion by coincidence: there are no relationships between the various parts of the component, they are simply gathered in this component.
- Logical association: the components which carry out similar operations are gathered in only one component.
- Temporal cohesion: all the components which are activated at the same time are gathered. Procedural cohesion: the elements of a component constitute only one sequence of control.

None of these kinds of cohesion appear in our description because the idea of microarchitecture is to discover and join a set of classes related by a service that they must provide working together. The microarchitecture is modelled with hierarchical composition [All97] of classes where the idea is to see the microarchitecture as a black box. We have an entrypoint for the input and an entrypoint for the output usually represented by ports of components that are used as the interfaces of the microarchitecture.

- Cohesion of communication: all the elements of a component operate on the same input data and produce the same output data.

This kind of cohesion does not appear in the microarchitecture because all the components process the data producing different results, not necessarily related or of the same type.

- Sequential cohesion: the output of an element of the component is used in input of another element.

This kind of cohesion is shown in the microarchitecture because the components in the microarchitecture are connected and coordinated by the communication provided by the connectors mostly in a sequenced way. Thus, they are data transformers in a sequential way.

- Functional cohesion: each part of the component is needed to do only one function.

This kind of cohesion is expressed in the second developed algorithm. When we classify the classes we do it analyzing what kind of services it provides. Thus we distinguish between processing, data, connecting, controller, coordinator, interface, service provider, information holder or structuring components.

- Cohesion of object: each operation offers a functionality based on the modification, the inspection or the use of the attributes of the object.

This kind of cohesion can be seen in each component and it appears when we replace the non-deterministic choices by the use of an abstract machine B to show that the behaviour of a component is dependent of an internal state of the component.

Coupling, on the other hand, measures the interconnection between the various modules of a system. It is recommended to have lowest coupling allowing a greater independence of the

modules. In this case, we can evaluate the coupling at component-level, because we will take into account the communication between them provided by the connectors.

[San97] again identified seven levels of coupling:

- No direct coupling: two modules are without direct coupling if they correspond to two modules without any link.

This kind of coupling appears clearly in the description, because we only relate classes if they have one of the three proposed relationships in the mapping. Given two classes A and B, instances of B can be instances/class variables of A; instances of B used as parameters in one method of A; or instances of B are connected to an instance of A by a dependency mechanism.

- Coupling by data: the system whose information is represented internally in the components only communicate themselves by sending elementary data.

This kind of coupling appears in the description because Wright ADL does not allow us make inferences over structures of data. Thus, we must work with elementary data.

- Coupling by structured data: This coupling is an alternative of the coupling by data. This time they are structures of data which are sent by a module interfaces.

As we explained previously, this kind of coupling does not appear in the description.

- Coupling of control: it is a well-known coupling in the design of architectures. This coupling is observed when the internal execution of a module depends on the value of a variable called " flag " which is provided by another module.

This kind of coupling can be considered if we think that observing events in a component are 'flags'. Supposing this, when that event is executed in another component (in this second component it is an initiating event), this action shoots the starting process indicated by the observing event.

- External coupling: the coupling happens when modules are bound by an external environment with the software, for example, a communication protocol.

This kind of coupling is essential but must be limited to a restricted number of modules. We can say nothing about this kind of coupling because we focus our study to the internal parts of a framework and we do not find a framework with any external environment.

- Common coupling: the modules are rather independent from/to each other but have strong interconnections by global variables.

This kind of coupling is not fulfilled by the architectural elements, because the protocols defined in the ports of the components are related to the protocols of the roles of the connectors. There is no independency among the components and the connectors that represents a relationship between classes. Besides this, there is no global data.

- Coupling by contents: this mode of coupling takes place when a module uses information of data or control kept internally in another module.

This kind of coupling exists in our architectural description but considering that the communication between the components is made through events and they can have parameters as data.

The considerable advantage to have strongly coherent components and slightly coupled is that these components are well defined and independent. That thus enables us to more easily carry out the maintenance and the reusability of a component.

6.13 Architectural Views

The goal of a software architecture is to address each concern in order to satisfy all the expectations of the various stakeholders involved. With respect to this goal, a software architecture incorporates many different views of the system. These views are complementary and allow to improve the understanding of the system showing various aspects of it [LC98]

A non-exhaustive list of the views which are more commonly developed can be found in [GCBA95]. Among them, we just keep the views which are possible to get with our approach:

- ⇒ Structural/topological view : this view describes the components and the connections of the system and the topology of the connected set of components. Since this topology may evolve dynamically, this view must show all the potential components and connectors. This view is obvious, because a configuration of an architectural description is a topological configuration in Wright ADL. The result is a topology of the classes and its interrelationships mapped as components and connectors respectively.
- ⇒ Behavioural view : this view describes the scheduling of system actions or system state transitions. This view is reflected by coordination of events made by the connectors to order the communication among the classes.
- ⇒ Growth view : this view describes maintainability, extensibility, adaptability, portability, scalability, and product line applicability of a system. This view can be seen when the architectural description grows through the modification/addition/grouping of components and connectors.

6.14 Complementariness with other Techniques

In the previous sections, we enumerate different problems and insufficiencies appeared using the Wright ADL to get an architectural description of a framework (component): lack of expressiveness in structural features, lack of expressiveness of timing and no possibility to express dynamic binding or creation/destruction of components. In fact, our approach is focused in using the formal approach given by an ADL to have a complementary technique to the existing ones. The main idea is combine structural and behavioural software descriptions. The structure describes how the different parts in a software system are arranged, and the behaviour of a system describes the way in which it functions or operates. According to [SPL98], objects are dynamic entities, changing their state and behaviour throughout their existence. Object-oriented design models such as UML (Unified Modelling Language) [UML], provide several diagrams for capturing the static and dynamic aspects of a particular domain. While object structure is often simple to model, object behaviour is extremely complex and requires numerous models to decompose the complexity. UML includes several diagrams for capturing the dynamic aspects of behavior, including statechart, activity, and interaction (sequence and collaboration) diagrams. The interaction diagrams tend to be flow-oriented, depicting primarily interobject message flow (sequence diagrams) and data flow (collaboration diagrams). The statechart and activity diagrams focus on modeling intra-object behavior, depicting the variations of behaviour an object exhibits during its life (statechart diagrams) and the internal operations of a particular class method or operation (activity diagrams).

If we intend to make a parallel work between these techniques and our approach to infer architectural description, we show different aspects that can be complemented from both sides: informal and informal approaches. Let's enumerate some possibilities.

The static aspects of the model can be described using the Wright ADL expressing the classes and their relationships, but also building class hierarchies in parallel when we need to change a class (mapped as a component) by a subclass (mapped as another component). Thus, we avoid to lose the idea of hierarchical relationship between the classes.

The object behaviour is localized in the port of a component. In fact, it is the same set of methods that appears in a class definition, but as they are classified in initiating/observing events in the architectural description, we know which services are initialized by the object and which ones the object must wait to be notified by another object to start them. Thus, we also have a way of complementing a class definition having extra information about the methods.

The computation part of a component also gives us local information about the sequence of messages that are executed in the methods bodies of the object. This aspect is more oriented to intra-object information, but if we are interested to know the communication between two classes that are not related directly, we should use collaboration diagrams. This notation offers us a way to see a set of classes related by methods calls in a global overview. For example, the different behaviours of the subclasses of the UpdateStrategy class to notify the sensor when it should

update its data were expressed in three different collaboration diagrams in the example shown in the Appendix A. This is a limitation because a collaboration diagram is started by only one message and each possible variant in one communication between two classes changes the diagram. Anyway, this kind of diagram offers a good panorama of communication between a set of classes.

The statechart and activity diagrams focus also on intra-object information. The different states of an object and how they behave based on a set state are expressed in non-deterministic choices using Wright ADL. Both techniques are complementary, one offers a visual idea and the other one offers the formal representation of the situation.

Another description technique that can be complemented by components described with Wright ADL is CRC Cards. In the CRC cards a class is described informally detailing its responsibilities and its collaborators. As our approach defines a direct mapping from classes to components/connectors, the CRC cards could add a section to represent the formal model of the class specifying how the different responsibilities are carried out in the computation of the architectural element.

Examples can also find a formal perspective with architectural topologies described with Wright ADL. Examples show particular functionalities of a framework through the use of a partial instantiation. Restricting the set of events of the involved components to the example, we can provide the topology corresponding to the particular instantiation

Summarizing, if we are interested in a complete description of a framework with complementary methodologies, we can consider the following viewpoints:

- ⇒ Global overview of the system structure: Components and Connectors
- ⇒ Objects with structural features : UML diagrams (e.g. Composite Pattern)
- ⇒ Interfaces of the objects : Port of the Components
- ⇒ Localized behaviour: Component Description
- ⇒ Relationship between two Objects (not necessarily connected in the Wright description): Collaboration diagrams

6.15 Summary

This chapter presents some general aspects that show the flexibility and the complementariness of the studied approaches. We also enumerate some advantages and disadvantages of using an ADL such as Wright: potential of the connectors, increment of components and connectors, management of errors, lack of expressiveness in timing and structural features, the non-possibility of expressing creation-destruction of objects and processes, etc. Thus, we get first analysis before considering the evolution aspect. This is an important point in which we are interested that can produce changes in the frameworks and we would like to see the impacts in the inference architecture.

Chapter 7: Evolution of Frameworks

An object oriented framework is defined as a reusable design of all or part of a system that is represented by a set of abstract classes and the way their instances interact or the skeleton of an application that can be customized by an application developer. The second definition comprises the first one, because we consider that the skeleton of the application is the abstract classes and the customization is the process of instantiation. Thus, the second definition gives the main feature of the framework: the customization of a design and code to get different applications in a specific domain. To get this level of a stable design and code, there was a development process that can be summarized with the following steps [Mat96]:

- Analysis of the problem domain
- The first version of the framework is developed using the key abstractions found
- One or possibly a few applications are developed based on the framework
- Problems when using the framework in the development of the applications are captured and solved in the next version of the framework
- After repeating this cycle a number of times the framework has reached an acceptable maturity level, and can be released for multiuser reuse in the organization

Following the approach of an iterative process, in chapter 5 we presented two algorithms to get an architectural description of an object oriented framework or a framework component in different levels of abstraction (going from a general behaviour to specific behaviour). Thus, we obtained an “architectural vision” or an interaction architecture [Now99] that shows us how the framework or the component can be seen in a ‘high-level’ of abstraction, with:

- classes and interrelationships mapped as components and connectors
- (in the second algorithm) assigning a semantic function to the classes in the framework.

Thus, we can have a classification of classes according to the kind of service that they provide, and not only a set of classes without any semantic relevance. For example, Policy class is a class which is in charged of policy access to a shared resource. In the studied model, this class is clearly mapped as a connector. However, not all the classes offer one single function. This situation happens mostly in object oriented frameworks already implemented (the case study of the first algorithm). In these cases, we adopt the idea of separating the internal state and the behaviour of the class in component and a connector. Thus, we focus the information on the component and all the management of communication in the connector.

But the requirements in the different domain applications can vary and it is necessary to update the framework to be able to instantiate it and cover all the new functions required. At this level, we talk about this phase as framework evolution. In this chapter we explain what are the possible changes that a framework can have and how they affect to the architectural description that we proposed previously

7.1 General Overview of the Levels of Abstraction

During all the development of this dissertation, we have found that most of the literature considers the view shown in the figure 7.1 to speak about different levels of abstractions. This view is the original conception of a system, where a top-down methodology was applied to build a system. Considering this viewpoint, from one perspective software evolution can take place at three fundamentally different abstraction levels: the implementation level, the design level, and the architectural level. Depending on the characteristics and extent of an implementation change, it may result in no major design changes, and thus no major architectural changes. However, it may also result in radical design and architectural changes. Changing the software design, may result in

no major changes. For example, if the change can be realized simply by reorganizing parts of the old version in new version. Most likely however, a change to the design could also imply major changes on both the architectural and implementation level. Finally, a change of architecture typically result in major changes of the design and implementation. In very special cases, we can imagine a change of architecture not resulting in major design and implementation changes. However, these must be considered exceptional cases.

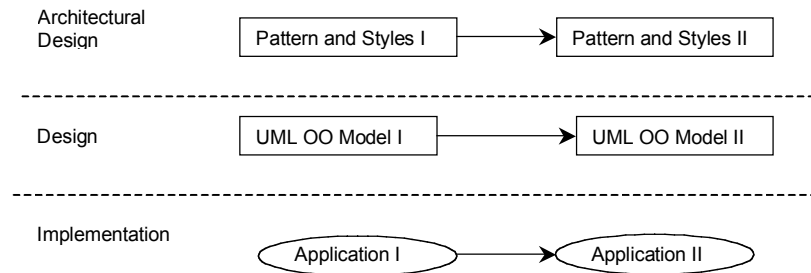


Figure 7.1: Software Evolution at Three Levels of Abstraction

But the problem proposed in this dissertation is that once the system was built, the documentation of all the design decisions which can reflect the design and architectural level (proposed by [Now99]) is expressed informally and using the line-box diagrams. Thus the changes in the framework are very difficult to trace in the documentation. The final result is that the changes are only made in the code, new documentation is generated and we have versioning of the application and the documentation. Thus, detecting a change in the code implies to run tools over different versions of the application and compare them to show the changing parts of the application. However, our proposal is based on a recovering an 'architectural description' of a built object oriented framework based on the design and code and use it as a documentation source for future changes. Our architectural description is very tied to the source code and so any change will have an impact on it. Next subsection shows briefly a discussion of possible changes in a framework. Based on them, we search to see how to track the evolution issue, the use of rules to reflect the changes and evaluate the quality of the proposed architectural description.

7.2 Aspects of Evolution

Using object oriented frameworks as components and working in the software product-line architecture and the effect of new or changed products requirements, [Mat00a] identifies a classification scheme of framework evolution. In fact, this proposal works in a coarser grained level of components than one considered in this dissertation but the different features of evolution are applied at framework level. Thus, we show the proposed categorization and analyze how these changes are (not) propagated in the architectural description proposed in the previous chapter.

7.2.1 Internal reorganization

In response to new requirements, the framework may be reorganized internally without affecting the externally visible functionality of the framework. The reason for reorganization is to improve one or more of the quality requirements (sometimes referred to a non-functional requirements) of the framework, e.g. flexibility, reusability or performance.

Hot spot introduction: The client of the framework has a possibility to use this new flexibility option or rely on the original default behaviour. Introduction of a new hotspot generally improves the flexibility quality attribute of the product.

In application frameworks, we take only into account three possibilities [FPR00]: introduction of an abstract method in an existing class, introduction of a new class and introduction of a set of classes (microarchitectures).

Let's see step by step how we must change our description in each case.

Introduction of an abstract method in an existing class:

1. Transform the new method in a process
2. Detect the component which represents the class.
3. Add as parameter the method mapped as a process.
4. Adapt the protocol of the component in the ports and in the computation with the new process. This can mean add the process in the right position in the chain of execution.
5. Change the roles of the connector that is affected by the previous change.
6. If the change is located in the first levels of description, propagate the changes in the successive descriptions

Introduction of a new abstract class

1. Model the class as a component considering the behaviour as initiating/observing events and transform the abstract method(s) in process(es).
2. Detect the classes (and the respective components) affected by the relationship with the new class.
3. Modify the connectors with the new relationships and the new protocols.
4. If necessary, modify the affected components. In some cases, the components can have a new task based on the event of the new component.
5. If the change is located in the first levels of description, propagate the changes in the successive descriptions
6. If we have subclasses of the affected component, we should try to test the change in the different level of description considering this new subclass

Introduction of a new microarchitecture

In this case, we must run the first algorithm to build the new microarchitecture (including following the suggested steps for modelling abstract methods or classes).

Following the same steps for the introduction of a new abstract class (considering all the microarchitectures as the new class) considering which are the entry points for the microarchitecture.

Restructuring: It may be necessary to perform some framework restructuring using refactoring techniques [Opd92] to comply to new requirements. An example of a possible refactoring is transforming a class hierarchy relationship into an aggregate relationship. These kinds of transformations are behaviour preserving but result in a changed reuse interface of the framework. Restructuring is often used for achieving better maintainability of the framework and only affects the internal structure of the framework since the restructuring is behavior preserving.

Let's see two of some examples of refactoring proposed in [DDN99] and how these changes can be shown in the architectural description. Most of them look for creation/modification/removal of a superclass/subclasses together with a number of pull-ups or push-downs of methods and attributes: Split into Superclass/Merge with Superclass, Split into Subclass/Merge with Subclass, Move to other Class (Superclass, Subclass or Sibling Class), Split Method/Factor Out Functionality

Split into Superclass

1. Detect the components which must be splitted.
2. Classify the (set of) events that must be distributed in the new three components.
3. Build the new three components with their respective protocols in the ports and the computation.
4. Detect the original connectors of the components.
5. If necessary, build new connectors with the subset of events of the protocols.

6. If the change is located in the first levels of description, propagate the changes in the successive descriptions

Merge with Superclass

1. Detect the components which must be merged.
2. Classify the (set of) events that must be distributed in the two components, which will have the final result.
3. Build the two components adapting their respective protocols in the ports and the computation.
4. Detect the original connectors of the component to be removed.
5. Adapt the connectors of the new two components with the new events.
6. If the change is located in the first levels of description, propagate the changes in the successive descriptions

Changing control flow: in cases where the order of actions in response to an event needs to be changed, the flow of control between the objects in a framework needs to be changed. One may argue whether this is a change in functionality, since the same behavior is executed, although in a different order.

This change in the architectural description is focused on the connectors, because they coordinate the communication between the different classes mapped as components. So the change of control flow is the change in the sequence of events in the glue of the connectors that communicate the classes affected by this evolution issue. This aspect is not so critical, because the propagation in the different levels of description is just only the change of a protocol in a connector

Refining framework concepts: The new requirements may point out flaws in the framework design such as classes that has two roles that now have to be refined further. This action will make the framework more understandable and it may provide new hot-spots to be added to the reuse interface. Concept refining does not affect the internal behaviour but it does change the internal structure of the framework.

This issue can be considered as a mixture of refactoring and introduction of hotspots.

7.2.2 Changing functionality

The functionality of the framework may need to be changed. For instance, due to new underlying hardware the way the framework interacts with the hardware needs to be adapted. The calling interface and structure of the framework tend to remain constant, but the behavior in response to events is changed.

Changing the internal class behaviour : This behavioural change is localized to only one class in the framework and the class affected has no complex behavioral relationships to other classes.

This aspect just only affects the definition of a class, thus, we must change the respective component in the architectural description, and also propagate the changes if the change is made in the first levels of description. In spite that the definition says 'no complex behavioral relationships', the relationships of the classes are mapped on the connectors and the ports and the roles protocols are tied in a configuration definition through the attachments. Then we must propagate the change if some events are added or removed from the protocols definition in the component.

Restructuring the subsystem: When extending the product family with new products, a typical situation is where a framework needs to provide different versions of a part of its behaviour. Assuming that no preparations were taken in the original framework design, the version-dependent functionality will often be spread out over the framework. This requires a restructuring and

partitioning of the framework's behavior, generally leading to a division into two subsystems, containing version-independent and version-dependent behavior, respectively. The subsystems are often organized in a layered or master-slave like structure.

In this case, we have a large impact in the framework because we need to restructure and partition the complete framework to adapt them. This impact obviously affects the architectural description too. We recommend to identify changes in microarchitectures and propagate them in the architectural description to see gradually how the changes affect it. Another possibility is just generate all the changes and run the algorithms again, keeping as documentation the old and the new versions of the architectural descriptions.

7.2.3 Extending functionality

A very common type of framework evolution is the extension of the framework functionality. Especially when adding new products to a product family, additional functionality may be required from one or more of the reusable frameworks.

Functionality introduction: new requirements may demand the extension of the framework functionality. If the architectural requirement only affects one framework, the situation is similar to the traditional monolithic framework development process, which implies iterating the framework design for achieving a new framework version [Joh91] [Mat00b]. Our approach is focused on the architectural description of only one framework. Thus, the introduction of functionality can be seen as the introduction of a new microarchitecture and connect it to the existing classes. If so, we must build the architectural description of the microarchitecture and we must detect the entrypoint of the microarchitecture and which are the components and the connectors will be connected to the new part of the framework. Thus we must modify the protocols of the components and the connectors to adapt the new communication with the added microarchitecture.

Concept introduction: The introduction of new concepts in the framework causes the introduction of new functionality. New concepts require the reorganization of the framework since new concepts have relations to the existing concepts. Concept introduction often requires the use of earlier mentioned techniques, e. g. concept requirement, hot-spot introduction, etc.

7.2.4 Reducing functionality

In certain cases, functionality that was developed as an integral part of the framework functionality has to be removed from the framework. For example, one possible cause includes the introduction of a new specialized framework that requires the functionality it provides to be removed from the other frameworks.

Enable/disable introduction: there may be product family requirements that require the introduction of an enable/disable mechanism in the framework. This type of evolution is not so common compared to the rest of the types of evolution taking into account that we work with small frameworks. But the an enable/disable mechanism can be thought as the introduction of a new component that has the responsibility of manage this, or just the modification of protocols of the components and the connectors involved in the operation or simpler, the use of boolean variables.

In all the cases we must change (or create a new component and) adapt roles and ports protocols to reflect the new changes.

Cutting functionality: If new requirements for a product family require the use of an additional framework that has to collaborate with the existing framework, conflicts due to of domain overlap or implicit assumptions about ownership of the thread of control may occur (e.g. [MB00][SBC96]). This will make it necessary to cut functionality out of the existing framework component since it

may be not possible to make necessary changes in the new framework, for instance, due to lack of access of source code or the complexity.

Thus, to cut the functionality means the deletion of components and connectors. Thus, this action implies that firstly, we must detect which of the components are connected to the potential-non-existing components and check which are the events to be deleted in the affected components, and adapt the respective protocols with the reduced set of events.

7.3 Analysis

Except in the case of 'change of control flow', which is a change focalized in the connectors, the rest of the evolution aspects causes many changes in our set of architectural descriptions (obtained with several runnings of our algorithms). If the changes are located in the first levels of description (general aspects), it will also provoke a propagation in the rest of levels (specific aspects). But it is clear that this situation happens because, at the level where are working (close to code-level) and the size of object oriented frameworks are very small and our mapping relates tightly a class to a combination of a component-connector. Thus, the format of a component gives information about the class structure, the ports protocols says the functionality of the class and the connector shows how the different classes are related and how they collaborate to fulfil the services provided by the framework. So if we reflect a change in the structure of classes, we have clearly a change in a set of components and connectors (resulted from the mapping). In the best case, we localize a change in only one class and thus, a modification in component; but in the worst case, for example refactorings, the changes at structural level are deep in the architecture and must be propagated in all the levels of description. If we reflect a change in the functionality of classes, the changes are localized in the computation and glue protocols, where all the classes behaviour is expressed.

Considering that

- We are working at a low level of abstraction (code-level)
- An evolution in the framework changes the structure/functionality of the architectural description proposed by the dissertation
- We propagate the changes from the more generic to specific behaviour in the architectural description, because we are interested in keeping all the 'process of growth' as a documentation of the framework, complemented by other methodologies.

an evolution analysis helps to:

- ⇒ Gain an overview of the system evolution
- ⇒ Qualify the parts of the system in terms of their stability over the versions
- ⇒ Identify system components which could be reused in other systems
- ⇒ Identify the migration of components between systems parts, and
- ⇒ Identify components that have appeared or disappeared over the life-cycle of the system
- ⇒ Isolate the different 'regions' of the frameworks that change very often and thus recognise the level of criticality of the different parts of the framework.

7.4 Summary

Based on scheme about framework evolution, this chapter summarizes the different types of changes that a framework can have. We also provide a brief analysis in each case that shows us what kinds of changes are made in the architectural description.

Chapter 8: Conclusions and Future Research

8.1 Context

In the research described in this thesis we have investigated the applicability of formal approaches of software architectures (such as Wright ADL and architectural patterns) to get an architectural description of object oriented frameworks, and also analyze the impacts of evolution issues in the inferred framework architecture.

To fulfill with these requirements, firstly we start to study all the features of software architectures, architectural styles and patterns, and ADLs. Thus, we discovered that the most systems where these concepts are applied were not concerned with the object oriented paradigm. This means that the systems were not designed using an object oriented methodology nor language. In this context the use of ADLs was concentrated in recovering the properties of these systems and mapped them in an architecture. If the resulting architecture could comprise classical architectural styles (such as pipes and filters, layers, etc), this would include the 'added values' of properties of these styles.

Secondly, we investigated the characteristics of object oriented frameworks in general and we concentrated our analysis in application and components frameworks and how the informal documentation techniques describe what are the services provided by the frameworks and what is required to instantiate it for some application, for example. In this field, we found different approaches. Most of them were guided by a similar idea presented in this thesis: the combination and complement of several artifacts of domain analysis and object oriented design to give different views of a framework's architecture. Among these techniques we can mention usage scenarios, examples, patterns and contracts. But, one critical issue of these artifacts is that they do not allow us to make any inference about the formal properties about the framework that they describe because the semantics of the parts are lost in box-line drawings, and thus, we do not have information about design decisions taken during the building process and use of the framework. To get them, in the worst case, we must understand the framework at code-level.

Finally, we developed the approach presented in this dissertation. It comprises of a mapping from Java/Smalltalk code to CSP-notation and two algorithms that allows to get an architectural description expressed in components and connectors with Wright ADL and architectural patterns as a complement. This work was made in three stages. The first one was an analysis of the common points between software architectures and object oriented frameworks to find the gaps that we had to fill to fulfill the objective. Based on the first stage, the second one consisted in the definition of the artifacts (proposed in this thesis) and the applicability of them in two of all the studied cases. In that, the approach was more understandable and easy to follow. The third step was comprised of a posterior analysis of the strengths and weaknesses of our approach and also a discussion of impacts of change when the framework evolves.

Following we present the conclusions of our approach and some open research issues that were generated during the development of this work.

8.2 Conclusions

The lack of established rules to define how we can infer an architecture from an application in general and the treatment of software architectures in a 'high level of abstraction' made us to define the constraints of our work context. Therefore, we restrict the use of the term architecture to the semantic structure of an application, a static notion of software architecture, and we consider that what was important was to understand how the classes in a framework were related structurally and what the consequences of this structure are on the software system. Classes seldom perform useful behaviour by themselves; they often contribute to a given function by interacting with other objects. We focus our work on a lowest or base-level notion of software architecture that reflects the semantic structure of a software system: the code-level combined with information about design-level in object oriented frameworks.

Starting from the code and design of an object oriented framework where the information about semantic structure was almost hidden, we get an architectural description with the following characteristics:

- The classes mapped as a combination of a component and a connector allows us to obtain two models: units of computation (components) and units of communication (connectors). Thus, we separate and localize the internal state of the class from the information about which classes are connected with.
- The model of communication (set of connectors) provides us all the behaviour that appeared in the framework and makes us explicit the ordering method calls and the protocol of interaction between the different classes.
- A classification of the messages inside a class: Using the concept of initiating and observing events of Wright ADL, we were able to know which messages were dependent only of the class and which ones were called by other classes.
- We provide levels of abstraction that gives us the architecture from a generic to specific view of the different parts of the framework.
- The detection of microarchitectures encapsulating them in a hierarchical description provided by Wright ADL.

Through the use of architectural styles we expressed the fixed parts of a framework (hotspots). Therefore we got a formal model that characterizes all the applications resulting from the framework instantiation.

About the mapping, we obtained a language-independent definition to transform statements to CSP-notation. The considered statements are also present in procedure oriented languages, not only in object-oriented ones, thus it can be useful to infer information about application generated in those languages.

The run of the algorithm getting a sequence of architectural description gave us a parameter to measure different qualities of the framework.

One of the most important points is that the developed description technique can be used as a complement with existing ones, because this is a way of providing the 'bridge' between informal and formal approaches.

However, during the use of Wright ADL as our main tool we found different problems:

- The static nature of this language made us difficult to add a management of errors. To do this we must change the interaction protocol of the components and connectors and we lose the expressiveness of the protocols because the behaviour and the errors messages are mixed
- The hierarchical relationship between two classes is not explicit in the description. If we change a component from one level of description to another one, and the latter component represented a subclass of the class mapped as the first component, this relationship is lost.
- The dynamic binding is expressed as two possibilities using non-deterministic choice. This fact is related as the absence of this ability in the ADL.

8.3 Future Research

An automatic tool to make the mapping and to run the algorithms would help us to validate this approach and to evaluate object oriented framework larger than we presented here.

An analysis of the information that can provides us the test of formal properties given by Wright ADL. In fact, the verification of the formal properties using the Wright tool was only used as a way of verification if the built model was a valid description. But, we think that it is useful to study what other information we can obtain from this verification.

The possibility of working in a higher level using all the architecture obtained through the use of algorithms, focusing on the actual tendency of software.

We do not handle important issues such as performance, reliability, and security. For many frameworks finding notations that expose such properties can be useful.

The formal model is essentially flat. To come up with this problem, our model we propose to work in different dimensions and levels of description (from general to specific behaviour). It would have been much nicer to be able to mirror the inheritance structure in the architectural specification.

A formal representation of design patterns: in the case of component framework proposed by [CTN97], they apply a good principle of composition. They got adaptable components as black-box entities that provide a solution to coordination problems. One of the components encapsulates the Acceptor-Connector Pattern which provides a communication service. Based on this idea, it is interesting to explore which design patterns can be encapsulated as a component with plugs and we could generate a formal representation of it. Thus we would have not only the solution offered by the design pattern but also a formal representation with classical properties checked by the Wright tool.

Finally, it would be interesting to extend this work with ADL which allows dynamic architectures (where components appear and disappear, or where the topology of a configuration changes during an execution): such as PICCOLA, π -Space, dynamic Wright. Some languages may include all potential elements in the system description, and then ignore those that do not currently exist. Others, however, describe each possible configuration as a different architecture

References

- [AAG93] G. D. Abowd, R. Allen, and D. Garlan. Using style to understand descriptions of software architectures. *ACM SIGSOFT Software Engineering Notes*, 18(5):9-20, December 1993
- [Abr96] J-R. Abrial. *The B Book*. Cambridge University Press, 1996
- [Ada95] D. Adair. Building Object-Oriented Frameworks, *AIXpert*, February-May 1995
- [AG94] R. Allen and D. Garlan. Formalizing architectural connection. In *Proceedings 16th International Conference on Software Engineering*, pp. 71-80. IEEE Computer Society Press / ACM Press, 1994
- [AG96] R. Allen and D. Garlan. A Case Study in Architectural Modelling: The AEGIS System. In *Proceedings of 8th International Conference on Software Specification and Design (IWSSD-8)*, March 1996
- [AGI98] R. Allen, D. Garlan, and J. Ivers. Formal modeling and analysis of the HLA component integration standard. *ACM SIGSOFT Software Engineering Notes*, 23(6): 70-79, November 1998. Proceedings of the ACM SIGSOFT Sixth International Symposium on the Foundations of Software Engineering
- [Ale76] C. Alexander. *The Timeless Way of Building*. New York: Oxford University Press. 1976.
- [All97] R. Allen. *A Formal Approach to Software Architecture*. Ph. D. Thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, May 1997
- [And94] G. Andert. Object Frameworks in the Taligent OS. *Proceedings of Comcon 94*, IEEE CS Press. Los Alamitos. California. 1994
- [Ber98] L. Bergmans. A Notation For Describing Conceptual Software Architectures. In *Proceedings of the First Nordic Software Architecture Workshop (NOSA'98)*. University of Karlskrona/Ronneby. August, 1998
- [BGK+97] D. Baumer , G. Gryczan, R. Knol, C. Lilienthal, D. Riehle and H. Zullinghoven *Communications of the ACM*, 40(10): 52-59, October 1997
- [BJ94] K. Beck, R. Johnson. Patterns generate Architectures. In *Proceedings of the 8th European Conference on Object Oriented Programming*. Bologna. Italy. 1994
- [BMR+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerland, and M. Stal. *A System of Patterns – Pattern Oriented Software Architecture*. Wiley, 1996
- [Bos00] J. Bosch. Measurements Systems Frameworks. In *Domain Specific Application Frameworks: Frameworks Experience by Industry*. M. Fayad, R. Johnson editors, Wiley Press, 2000
- [BR96] T. Biggerstaff and C. Richter. Reusability framework, assessment , and directions. *IEEE Software* 4(2): 41-49, March 1996
- [Bro00] L. Bross. *Box Structures as an ADL*. Master Thesis to appear. University of South Florida, 2000. <http://home.tampabay.rr.com/adls/archben.html>
- [CDK94] G. Coulouris, J. Dollimore and Tim Kindberg. *Distributed Systems Concepts and Design*. Addison Wesley, 1994
- [CES85] E. M. Clarke, E. A. Emerson, and A. P. Sistla. *Automatic verification of finite state concurrent systems using temporal logic specifications*. Technical report, Austin, 1985
- [CTN97] J. C. Cruz, S. Tichelaar and O. Nierstrasz. A Coordination Component Framework for Open Systems. In *Proceedings of COORDINATION '97*, 1997
- [Deu89] L . P. Deutsch. Design Reuse and Frameworks in the Smalltalk-80 system. In *Software Reusability*, Vol II, T. J. Biggerstaff and A. J. Perlis, editors . ACM Press, 1989
- [DDN99] S. Demeyer, S. Ducasse and O. Nierstrasz. Finding Refactoring via Change Metrics. *Proceedings of OOPSLA 99*. 1999
- [DMNS97] S. Demeyer, T. D. Meijler, O. Nierstrasz, and P. Steyaert. Design guidelines for tailorable frameworks. *Communications of the ACM*, 40(10): 60-64, October 1997
- [FPR00] M. Fontoura, W. Pree and B. Lumpe. UML-F: A Modeling Language for Object-Oriented Frameworks. In *Lecture Notes in Computer Science Nr. 1850* Proceedings of ECOOP 2000 Elisa Bertino (ed.) Springer-Verlag. June 2000.
- [FSJ00] M. Fayad, D. Schmidt, and R. Johnson. Application Frameworks. In *Building Application*

- Frameworks: Object Oriented Foundations of Framework Design*. M. Fayad, D. Schmidt, and R. Johnson (editors). Wiley Press. 2000
- [GA094] D. Garlan, R. Allen, and J. Ockerbloom. Exploiting style in architectural design environments. In *Proceedings of SIGSOFT '94: The Second ACM SIGSOFT Symposium 1994 on the Foundations of Software Engineering*, December 1994
- [GA095] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software*, 12(6): 17-26, November 1995
- [Gar95] D. Garlan. What is Style ? In *Proc. First International Workshop Software Architecture*, April 1995
- [GCBA95] C. Gacek, B. Clark, B. Boehm, and A. Abd-Allah. On the definition of software system architecture. In *Proceedings of the 1st International Workshop on Architecture for Software Systems*, pp. 85-94, Seattle, WA, April 1995
- [GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Design*. Addison Wesley, 1995
- [GMW97] D. Garlan, R. Monroe, D. Wile: *ACME, An Architecture Description Interchange Language*, White-paper, 1997
- [Gol84] A. Goldberg. *Smalltalk-80: The Interactive Programming Environment*. Reading MA: Addison-Wesley. 1984
- [GP94] D. Garlan and D. Perry. Software architecture: Practice, potential and pitfalls. In *Proceedings: 16th International Conference on Software Engineering*, pp. 363-364. IEEE Computer Society Press / ACM Press, 1994
- [GP95] D. Garlan and Dewayne Perry. Introduction to the Special Issue on Software Architecture. *IEEE Transactions on Software Engineering*, 21(4): 269-274, April 1995
- [GS92] D. Garlan and M. Shaw. An introduction to software architecture. In *Advances in Software Engineering and Knowledge Engineering*, pp. 1-40, V. Ambriola and G. Tortora Editors. World Scientific Publishing Co. , 1992
- [HJE95] H. Hueni , R. Johnson and R. Engel. A Framework for Network Protocol Software. *Proceedings of OOPSLA 95*, Austin, Texas. October 1995
- [Hoa85] C. A. R. Hoare. *Communicating Sequential Programming*. Prentice Hall, 1991
- [Hol90] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1990
- [HR94] F. Hayes-Roth. *Architecture-based acquisition and development of software: Guidelines and recommendations from the ARPA domain-specific software architecture (DSSA) program*. Technical Report, Teknowledge Federal Systems, Palo Alto, CA, October 1994
- [Joh88] R. Johnson, B. Foote. Designing Reusable Classes, *Journal of Object-Oriented Programming*, Vol .1, No. 2, June 1992
- [Joh91] R. Johnson. *Reusing Object Oriented Design*, University of Illinois, Technical Report UIUCDCS 91-1696, 1991
- [Joh92] R. Johnson. Documenting frameworks using patterns. *ACM SIGSOFT Notices*, 27(10): 63-76, October 1992. OOPSLA '92 Proceedings, Andreas Paepcke (editor)
- [Joh93] R. Johnson. *How to Design Frameworks*, Tutorial Notes, 8th Conference on Object-Oriented Programming Systems, Languages and Applications, Washington, USA, 1993
- [KP88] G. Krasner and S. Pope. A cookbook for using the MVC user interface paradigm in Smalltalk-80. *Journal of Object Oriented Programming* 1(3):26-49. September 1988
- [LC98] P. Lalanda and S. Cherki. *Object oriented methods and software architectures*. Position Paper in ECOOP 98, Workshop on Object Oriented Software Architecture. 1998
- [LK94] R. Lajoie and R. Keller. Design and reuse in object-oriented frameworks: Patterns, contracts, and motifs in concert. In *Proceedings of the 62nd Congress of the Association Canadienne Française pour l'Avancement des Sciences (ACFAS)*, Montreal, Canada, May 1994.
- [LKA+95] D. Luckham, J. Kenney, L. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering*, 21(4): 336-355, April 1995
- [LT89] N. A. Lynch and M. R. Tuttle. *An introduction to input/output automata*. CWI Quarterly, 2(3): 219-246, 1989

- [LV95] D. Luckham and J. Vera. An event-based architecture definition language. *IEEE Transactions on Software Engineering*, 21(9): 717-734, September 1995
- [Mat96] M. Mattsson. *Object Oriented Frameworks: a survey of methodological issues*. Licentiate Thesis, Department of Computer Science, Lund University, 1996
- [Mac00] A. Maccari. *Architectural Evolution of product families*. Position Paper in ECOOP 2000 Workshop on Architectural Evolution. Cannes . 2000
- [Matt00a] M. Mattsson. *Object oriented Frameworks as Components – Experiences of Framework Evolution*. Position Paper in ECOOP 2000 Workshop on Architectural Evolution. Cannes . 2000
- [Matt00b] M. Mattsson. *Evolution and Composition of Object-Oriented Frameworks*. PhD Thesis. Department of Software Engineering and Computer Science. University of Karlskrona/Ronneby. 2000
- [MB00] M. Mattsson and J. Bosch. Object oriented frameworks: Composition problems, causes and, solutions. In *Building Application Frameworks: Object-Oriented Foundations of Framework Design*, pp. 467-487, M. Fayad, D. Schmidt, R. Johnson editors, Wiley Press, 2000
- [MDEK95] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures, *Lecture Notes in Computer Science*, 989, 1995
- [Men00] K. Mens. *Automating architectural conformance checking by means of logic meta programming*. PhD Thesis. Vrije Universiteit Brussel. June 2000
- [MK96] J. Magee and J. Kramer. Dynamic structure in software architecture. In *Proceedings of 4th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE 4)*, SEN, Vol.21, No.6, pp. 3-14, New York, October 1996. ACM Press
- [MKMG96] R. T. Monroe, D. Kompanek, R. Melton, and D. Garlan. Stylized Architecture, Design Patterns, and Objects. *IEEE Software*, Jan 1997, pp. 43-52
- [MN95] T. D. Meijler and O. Nierstrasz, Beyond Objects: Components, *Cooperative Information Systems: Current Trends and Directions*, M.P. Papazoglou and G. Schlageter (Eds.), pp. 49-78, Academic Press, November 1995
- [MORT96] N. Medvidovic, P. Oreizy, J. Robbins, and R. Taylor. Using object-oriented typing to support architectural design in the C2 style. *ACM SIGSOFT Software Engineering Notes*, 21(6): 24-32, November 1996
- [MQR95] M. Moriconi, X. Qian, and R. A. Riemenschneider. Correct architecture refinement. *IEEE Transactions on Software Engineering*, 21(4): 356-372, April 1995
- [MT97] N. Medvidovic and R. Taylor. A framework for classifying and comparing architecture description languages. In *Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE 97)*, pp. 60-76, Lecture Notes in Computer Science Nr. 1013, Springer-Verlag, September 1997
- [MTW96] N. Medvidovic, R. N.Taylor, and E. J. Whitehead, Jr. Formal modeling of software architectures at multiple levels of abstraction. In *Proceedings of the California Software Symposium*, pp. 28-40, 1996
- [Neb98] R. Nebbe. Semantic Structure: A Basis for Software Architecture. In Proceedings of the ECOOP '98 Workshop on Object-Oriented Software Architectures, *Lecture Notes in Computer Science Nr. 1543*, J. Bosch and S. Demeyer editors. Springer-Verlag, 1998
- [Now99] P. Nowack. *Interacting Components – A Conceptual Architecture Model*. Position Paper in ECOOP 99 Workshop on Architectural Evolution. Lisbon . 1999
- [Opd92] W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD Thesis. University of Illinois at Urbana-Champaign. 1992
- [Pet77] J. L. Peterson. Petri nets. *ACM Computing Surveys*, 9(3): 223-252, September 1977.
- [Pre95] W. Pree. *Design Patterns for Object-Oriented Software Development*. Reading, MA: Addison-Wesley, 1995
- [PW92] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4): 40-52, October 1992
- [Ric98] T. Richner. Describing framework architectures: more than Design Patterns. In Proceedings of the ECOOP '98 Workshop on Object-Oriented Software Architectures, *Lecture Notes in Computer Science Nr. 1543*, J. Bosch and S. Demeyer editors.

- Springer-Verlag, 1998
- [RJFC94] F. C. Ribeiro Justo and P. R. Freire Cunha. Deadlock-free configuration programming. In *Proceedings of the 2nd International Workshop on Configurable Distributed Systems*, March 1994
- [San97] R. Sanlaville. *Description d'architecture logicielles: Utilisation du formalisme Wright pour l'interconnexion de machines abstraites B. Report for DEA d'Informatique: Systems et Communications*. Laboratoire LSR (Logiciels, Systèmes, Réseaux). Université Joseph Fourier. Grenoble. France. 1997
- [Sch95] H. Schmid. Creating the Architecture of a Manufacturing Framework by Design Patterns. In *Proceedings of the 10th Conference on Object-Oriented Programming Systems, Languages and Applications Conference*, Austin, USA, 1995
- [SBC96] S. Sparks, K. Benner, C. Faris. Managing Object-Oriented Framework Reuse. *IEEE Computer*. pp. 53-61. September, 1996
- [SDK+95] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik. Abstractions for software and architecture and tools to support them. *IEEE Transactions on Software Engineering*, April 1995
- [SG96] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996
- [SG99] J. P. Sousa and D. Garlan. Formal modeling of the enterprise JavaBeans component integration framework. In *Proceedings of FM '99, Lecture Notes in Computer Science 1079*, Toulouse, France, September 1999. Springer-Verlag
- [SPL98] L. Seiter, J. Palsberg, K. Lieberherr: Evolution of Object Behavior Using Context Relations. *IEEE Transactions on Software Engineering*, 24(1): pp. 79-92 (1998)
- [Szy98] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley and ACM Press. New York, NY, 1998
- [Tal94] Taligent Inc. *Building Object-Oriented Frameworks*. A Taligent White Paper, 1994
- [TLPD95] A. Terry, R. London, G. Papanogopoulos, and M. Devito. *The ARDEC/Teknowledge architecture description language. Technical report*, Teknowledge Federal Systems, 1995
- [Tra93] W. Tracz. Parameterized programming in LILEANNA. *Proceedings of the ACM/SIGAPP Symposium on Applied Computing*, pp. 77-86, E. Keaton, K. M. George, H. Berghel, and G. Hedrick editors, Indianapolis, February 1993, ACM Press.
- [UML] Rational Rose. UML Notation Guide Version 1.1. September 1997. <http://www.rational.com/UML>
- [Ves96] S. Vestal. *MetaH Programmer's manual*, 1996
- [WBJ90] R. Wirfs-Brock, and R. Johnson. Surveying current research in object-oriented design. *Communications of the ACM* 33(9): 104-124, 1990

Appendix A

This appendix shows a complete analysis of the Application of Wright Architectural Description Language to describe a Framework. The objective is to see an step-by-step process of how to get an architectural description using this ADL

BRIEF DESCRIPTION OF THE MEASUREMENT SYSTEMS

Measurement systems are a class of systems to measure the relevant values of a process or product. It is used for quality control of parts entering production or of produced products that can then be used to separate acceptable from unacceptable items to categorize the products in quality categories.

A measurement system consists of more than sensors and actuators. A typical measurement cycle starts with a trigger, indicating that a product, or a measurement item, is entering the system. The first step after the trigger is the data collection phase performed by the sensors. The sensors measure the relevant variables of the measurement item. The second step is the analysis phase, during which the data from the sensors is collected in a central representation and transformed until it appears in a form in which it can be compared to the ideal values. Based on this comparison, certain discrepancies can be deduced, which in turn, lead to a classification of the measurement item and is used to perform associated actions, such as rejecting the item, which causes the actuators to remove the item from the conveyer belt and put it in a separate store, or to print the classification on the item so that it can be automatically recognized at a larger stage.

One of the requirements for the analysis phase is that the way the transformation takes place, the characteristics on which each classification should be flexible and easily adaptable during system construction, but also, to some extent during the actual system operation.

ARCHITECTURE OF THE SYSTEM

[Bos00] identifies five entities that communicate with each other to achieve the required functionality

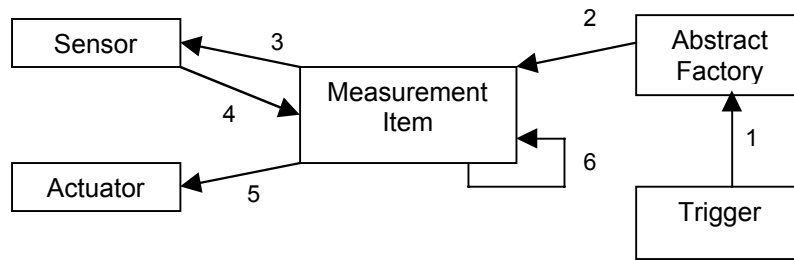
- The trigger triggers the abstract factory when a physical item enters the system
- The abstract factory creates representation of the physical object in the software, that is, the measurement item.
- The measurement item requests the sensor to measure the physical object
- The sensor sends back the result to the measurement item that stores the results
- After collecting the required data, the measurement item compares the measured values with the ideal values.
- The measurement item sends a message to the actuator requesting the actuation appropriate for the measured data.

The necessary details about the design and the implementation of the framework will be described later in order to understand the different decisions made in the architecture description.

After a first analysis made starting from the design documents [Bos00], it is possible to identify the following components: *Trigger*, *Item Factory*, *Measurement Item*, *Measurement Values*, *Sensor* and *Actuator*. We identified the same entities as the author, but we added also the Measurement Value because this component has an active interaction with Measurement Item.

So far, the described components are the main part of the Measurement Systems software, because in the most of the systems, a substantial part is hardware, since the system is connected to the real world through a number of sensors and actuators. Then we added the following components *Physical Trigger*, *Physical Sensor*, *Physical Actuator*

The figure 1 shows the studied architecture



FIRST LEVEL

In this proposal, if the component A communicates with the component B through the event m, one of the ports of the A will be called as B. Thus, in some way, we keep close to the implementation, because if an event in the computation has the following format

B.m is similar to a structure **object message**

B.m!x is similar to a structure **object message (parameters)**

B.m?x is similar to a structure **x _ object message**

B.m!x?y is similar to a structure **y _ object message (parameters)**

About the connectors roles, we will use the classnames that are connected by the connector

ANALYSIS

Based on the previous ideas, we will start an analysis of all the architecture following the process and identifying the participants and their interactions.

1. The trigger triggers the abstract factory when a physical item enters the system

Components : Physical Trigger, Trigger and the Abstract Factory.

Action: When a physical item enters the system, the physical trigger detects the entry and sends a signal to the trigger in the system. Then, the trigger triggers the abstract factory.

Components Specification

Component *C_Physical_Trigger*

Port *Out* = *T_Trigger_Out*

Computation = *Out.trigger* → *Computation _ Out.close* → §

Component *C_Trigger*

Port *In* = *T_Trigger_In*

Port *Item_Factory* = *T_Trigger_Out*

Computation = *In.trigger* → *Item_Factory.trigger* → *Computation*

_In.close → *Item_Factory.close* → §

Component *C_Item_Factory*

Port *Trigger* = *T_Trigger_In*
 ...
Computation = *Trigger.trigger* → ...
 _Trigger.close → ...

Connectors Specification

C_PhysicalTrigger →→ *C_Trigger* and *C_Trigger* →→ *C_Item Factory* uses the same connector

Connector *CN_Triggers*
Port *PTrigger* = *T_Trigger_Out*
Port *Trigger* = *T_Trigger_In*
Glue = *PTrigger.trigger* → *Trigger.trigger* → **Glue**
 PTrigger.close → *Trigger.close* → §

Observation: Physical Trigger and Trigger have similar behaviours, but there is one little difference given in the non-deterministic or deterministic choices of **close** event. In Physical Trigger is non-deterministic, because when there are no more items in the control process, this component can decide to finish the software execution. The Trigger must wait the notification of closing the system from the Physical Trigger, and thus to notify the rest of the components.

2. The abstract factory creates representation of the physical object in the software, that is, the measurement item.

In the design [Bos00], the Item Factory incorporates both the Abstract Factory and the Prototype design patterns [GHVJ95] and it is responsible for instantiating instances of class Measurement Item whenever it receives a trigger event, to configure these instances and to activate each instance by providing it with a separate process or invoking its start method.

Components: Item Factory and Measurement Item

Action: When the Item Factory is notified by the Trigger, the Item Factory starts a new Measurement Item choosing between if the measurement item will (or not) be in calibration state.

Components Specification

Component *C_Item_Factory*
Port *Trigger* = *T_Trigger_In*
Port *Measurement_Item* = *T_CalibrationState_Out*
Computation = *Trigger.trigger* →
 (*Measurement_Item.calibrate* → *Measurement_Item.start* → **Computation**
 _Measurement_Item.normal → *Measurement_Item.start* → **Computation**
 _Trigger.close → *Measurement_Item.close* → §

Component *C_Measurement_Item*
Port *Item_Factory* = *T_CalibrationState_In*

Computation = *ItemFactory.calibrate* → *Item_Factory.start* → ...
 Item.Factory.normal → *Item_Factory.start* → ...
 _In.close → ...

Connectors Specification

Item Factory $\rightarrow\rightarrow$ Measurement Item

Connector CN_ToMI

Role $\overline{IF} = T_Calibration_Out$

Role $MI = T_Calibration_In$

Glue = $(\overline{IF.calibrate} \rightarrow \overline{MI.calibrate} \quad \overline{IF.normal} \rightarrow \overline{MI.normal})$

$\rightarrow \overline{IF.start} \rightarrow \overline{MI.start} \rightarrow \mathbf{Glue}$

$\quad \overline{IF.close} \rightarrow \overline{MI.close} \rightarrow \S$

Observation : In the Computation of $C_Item_Factory$, there is a non-deterministic choice between two events *CALIBRATE* and *NORMAL* and this is because of the class structure of Item Factory contains an instance of MeasurementItem and a state variable indicating whether the system is in calibration mode or in normal operation [Bos00]. The state is initialized in the Item Factory, thus, we can think that the choice between the two different modes of operation has to be made by itself.

3. The measurement item requests the sensor to measure the physical object

The measurement item is the object that contains the data collected from the sensors concerning the physical measurement item and the Measurement Value represents one aspect of the physical measurement item that is used by the system [Bos00]. Thus, the measurement item has a collection of measurements values, and these last ones represent the data of the measurement item.

Components: Measurement Item, Measurement Values and Sensor

Action: When the measurement item requests the sensor(s) to measure the physical object, in fact, this action is forwarded to the measurement values of the measurement item, and these components will get the value from the connected sensors. In this stage, the calibration/normal state is not important, so in both states the calculation phase is made.

*Components Specification***Component** $C_Measurement_Item$

Port $Item_Factory = T_CalibrationState_In$

Port $MValues = T_Calculation_Out$

.....

Computation = $ItemFactory.calibrate \rightarrow Item_Factory.start \rightarrow \mathbf{Calculate} \rightarrow \dots$

$Item.Factory.normal \rightarrow Item_Factory.start \rightarrow \mathbf{Calculate} \rightarrow \dots$

$\quad \overline{In.close} \rightarrow \overline{MValues.close} \rightarrow \dots$

where $\mathbf{Calculate} = \overline{MValues.perform_calculation}$

Component $C_MeasurementValue$

Port $Measurement_Item = T_Calculation_In$

Port $Sensors = T_Measure_Out$

Computation = $Measurement_Item.perform_calculation \rightarrow \overline{Sensors.get_value} \rightarrow \dots$

$\quad \overline{Measurement_Item.close} \rightarrow \overline{Sensors.close} \rightarrow \S$

Component C_Sensor

Port $Measurement_Item = T_Measure_In$

.....
Computation = $\overline{\text{Measurement_Item.get_value}} \rightarrow \dots$
 $\underline{\text{Measurement_Item.close}} \rightarrow \dots$

Connectors Specification

Measurement Item $\rightarrow \rightarrow$ Measurement Values

Connector *CN_ToMV* (*numMvs*: 1..)
Role *MI* = *T_Calculation_Out*
Role *MV*_{1..numMvs} = *T_Calculation_In*
Glue = $\overline{\text{MI.perform_calculation}}$
 $\rightarrow \forall i:1..numMvs \ \underline{\text{MV}_i.\text{perform_calculation}} \rightarrow \mathbf{Glue}$
 ...
 $\underline{\text{MI.close}} \rightarrow \forall i:1..numMvs \ \underline{\text{MV}_i.\text{close}} \rightarrow \S$

Measurement Values $\rightarrow \rightarrow$ Sensors

Connector *CN_ToSensor* (*numSs*: 1..)
Role *MV* = *T_Measure_Out*
Role *Sensors*_{1..numSs} = *T_Measure_In*
Glue = $\overline{\text{MV.get_value}} \rightarrow \forall i:1..numSs \ \underline{\text{Sensors}_i.\text{get_value}} \rightarrow \dots$
 $\underline{\text{MV.close}} \rightarrow \forall i:1..numSs \ \underline{\text{Sensors}_i.\text{close}} \rightarrow \S$

4. The sensor sends back the result to the measurement item that stores the results

The Sensor is the software representation of a hardware device that measures one particular variable of the item in question. The sensor is responsible for maintaining an accurate model of the hardware sensor. To achieve this, the sensor communicates with the hardware sensor. The way the sensor updates itself with the data in the physical sensor can vary, depending on the sensor type and the application.

Components: Measurement Values, Sensor and Physical Sensor

Action: Following from the previous step, the sensor must send the measured value to the measurement item.

Components Specification

Component *C_MeasurementValue*
Port *Measurement_Item* = *T_Calculation_In*
Port *Sensors* = *T_Measure_Out*
Computation = $\overline{\text{Measurement_Item.perform_calculation}}$
 $\rightarrow \overline{\text{Sensors.get_value}} \rightarrow \text{Sensors.receive?val} \rightarrow \mathbf{Computation}$

 $\underline{\text{Measurement_Item.close}} \rightarrow \underline{\text{Sensors.close}} \rightarrow \S$

Component *C_Sensor*

Port *Measurement_Item* = *T_Measure_In*
Port *Physical_Sensor* = *T_Output_Input*
Computation = $\overline{\text{Measurement_Item.get_value}} \rightarrow$
 $(\overline{\text{Measurement_Item.send!val}} \rightarrow \mathbf{Computation})$
 $\underline{\text{Physical_Sensor.ask}} \rightarrow \text{Physical_Sensor.receive?result} \rightarrow$

$$\frac{\overline{\text{Measurement_Item.send!result}} \rightarrow \mathbf{Computation}}{_ \text{Physical_Sensor.ask} \rightarrow \text{Physical_Sensor.receive?result} \rightarrow \mathbf{Computation}} \\ _ \text{Measurement_Item.close} \rightarrow _ \text{Physical_Sensor.close} \rightarrow \S$$

Component $C_Physical_Sensor$

Port $Sensor = T_InputOutput$

Computation = $Sensor.receive \rightarrow \overline{Sensor.send!measure(val)} \rightarrow \mathbf{Computation}$
 $_Sensor.close \rightarrow \S$

Connectors Specification

Measurement Value $\rightarrow\rightarrow$ Sensor

Connector $CN_ToSensor (numSs: 1..n)$

Role $MV = T_Measure_Out$

Role $Sensors_{1..numSs} = T_Measure_In$

Glue = $MV.get_value \rightarrow \forall i:1..numSs _ Sensors_i.get_value \rightarrow$

$\forall i:1..numSs _ Sensors_i.send?val \rightarrow process_values \rightarrow \overline{MV.receive!val} \rightarrow \mathbf{Glue}$

$MV.close \rightarrow \forall i:1..numSs _ Sensors_i.close \rightarrow \S$

Physical Sensor $\rightarrow\rightarrow$ Sensor

Connector $CN_Sensors$

Role $Sensor = T_Output_Input$

Role $PSensor = T_Input_Output$

Glue = $Sensor.ask \rightarrow \overline{PSensor.receive} \rightarrow PSensor.send?result$

$\rightarrow \overline{Sensor.receive!result} \rightarrow \mathbf{Glue}$

$_Sensor.close \rightarrow \overline{PSensor.close} \rightarrow \S$

Observation : Firstly, it is supposed that the sensor stores the values sent by the physical sensor. We know that there is no way to store internal state of the components with a Wright description.

In the Component C_Sensor , the behaviour is given by different non-deterministic choices. The first one is given by the event ASK and the second one is when it has to return the measured value of an item aspect. As we explained before, the sensor updates itself, and this can be made for example, after a time interval the sensor updates its value asking the physical sensor, or when there was a change in the physical item, the physical sensor notifies the changed value to the sensor. Another way of updating is when it is required by the client of the sensor, in this case the measurement item. Thus, let's make a small analysis of the given non-deterministic choices. The first one is given by the need of updating the value asking the physical sensor (because of time interval). The second one is given when the measurement value ask the measured value. In this case, the sensor can send the store value, or ask the physical sensor to update the value and send it.

5. **After collecting the required data, the measurement item compares the measured values with the ideal values.**

Each measurement value contains a set value representing the correct value, a measured value representing the value measured at the current physical item, and a compare method describing how to interpret differences between the measured and set values

Components: Measurement Item, Measurement Values

Action: Based on the obtained data, in normal state of the system, the measurement item must decide if the physical item is acceptable or not in the production process. This is done comparing the measured values with the ideal ones and is performed by each measurement values because they keep the ideal and the measured values. Thus the process of comparison is forwarded to each measurement value, and if all answer positively, the item is acceptable, if not, the item is not acceptable.

Components Specification

Component $C_Measurement_Item$

Port $Item_Factory = T_CalibrationState_In$

Port $MValues = T_Calculation_Out$

.....

Computation = $ItemFactory.calibrate \rightarrow Item_Factory.start \rightarrow \mathbf{Calculate} \rightarrow \dots$
 $Item.Factory.normal \rightarrow Item_Factory.start \rightarrow \mathbf{Calculate}$
 $\rightarrow \mathbf{Compare} \rightarrow \mathbf{Computation}$
 $_In.close \rightarrow MValues.close \rightarrow \dots$

where $\mathbf{Calculate} = \overline{MValues.perform_calculation}$

$\mathbf{Compare} = \overline{MValues.compare} \rightarrow$

$MValues.acceptable _ MValues.non_acceptable \rightarrow \dots$

Component $C_MeasurementValue$

Port $Measurement_Item = T_Calculation_In$

Port $Sensors = T_Measure_Out$

Computation = $Measurement_Item.perform_calculation \rightarrow \overline{Sensors.get_value}$
 $\rightarrow Sensors.receive?val \rightarrow \mathbf{Computation}$

.....

$_Measurement_Item.compare \rightarrow$

$(\overline{Measurement_Item.acceptable} \rightarrow \mathbf{Computation}$

$_ \overline{Measurement_Item.non_acceptable} \rightarrow \mathbf{Computation})$

$_ Measurement_Item.close \rightarrow Sensors.close \rightarrow \S$

Connectors Specification

Measurement Item $\rightarrow\rightarrow$ Measurement Values

Connector $CN_ToMV (numMvs: 1..n)$

Role $MI = T_Calculation_Out$

Role $MV_{1..numMvs} = T_Calculation_In$

Glue = $MI.perform_calculation$

$\rightarrow \forall i: 1..numMvs _ \overline{MV_i.perform_calculation} \rightarrow \mathbf{Glue}$

.....

$$\begin{aligned}
_MI.compare &\rightarrow \overline{\forall i:1..numMvs _ MV_i.compare} \rightarrow \\
&(\forall i:1..numMvs _ MV_i.acceptable \rightarrow \mathbf{Glue} \\
&\quad \exists i:1..numMvs _ MV_i.non_acceptable \rightarrow \mathbf{Glue}) \\
_MI.close &\rightarrow \overline{\forall i:1..numMvs _ MV_i.close} \rightarrow \S
\end{aligned}$$

Observation: As we said in the previous step, it is also supposed that the measurement value keeps the last measured value and we explained that this was not possible. Thus, when the measurement value receives the event of comparing, the decision of acceptability is described as non-deterministic because in the actual description we do not have elements to evaluate this condition.

6. **The measurement item sends a message to the actuator requesting the actuation appropriate for the measured data.**

Components: Measurement Item, Measurement Values, Actuators and Physical Actuators

Action: In the normal state of the system, if the item is qualified as non-acceptable, the actuators must work. So the measurement item asks the actuators to work. But in calibration state, the measurement item asks the calibration to the measurement values themselves. This process consists in storing the real measured values as ideal values

Components Specification

Component C_Measurement_Item

Port $_Item_Factory = T_CalibrationState_In$

Port $MValues = T_Calculation_Out$

Port $Actuators = T_Actuate_Out$

Computation = $_ItemFactory.calibrate \rightarrow _Item_Factory.start$

$\rightarrow \mathbf{Calculate} \rightarrow \overline{MValues.calibrate} \rightarrow \mathbf{Computation}$

$_Item.Factory.normal \rightarrow _Item_Factory.start$

$\rightarrow \mathbf{Calculate} \rightarrow \mathbf{Compare} \rightarrow \mathbf{Computation}$

$_In.close \rightarrow \overline{MValues.close} \rightarrow \overline{Actuators.close} \rightarrow \S$

where $\mathbf{Calculate} = \overline{MValues.perform_calculation}$

$\mathbf{Compare} = \overline{MValues.compare}$

$\rightarrow \overline{MValues.acceptable _ MValues.non_acceptable}$

$\rightarrow \overline{Actuators.actuate}$

Component C_MeasurementValue

Port $_Measurement_Item = T_Calculation_In$

Port $Sensors = T_Measure_Out$

Computation = $_Measurement_Item.perform_calculation \rightarrow \overline{Sensors.get_value}$

$\rightarrow \overline{Sensors_i.receive?val} \rightarrow \mathbf{Computation}$

$_Measurement_Item.calibrate \rightarrow \mathbf{Computation}$

$_Measurement_Item.compare \rightarrow$

$\overline{(_Measurement_Item.acceptable \rightarrow \mathbf{Computation})}$

$_Measurement_Item.non_acceptable \rightarrow \mathbf{Computation})$

$_Measurement_Item.close \rightarrow \overline{Sensors.close} \rightarrow \S$

Component *C_Physical_Actuator***Port** *Actuator* = *T_Signal_In***Computation** = *Actuator.receive* → *actuate* → **Computation** *_Actuator.close* → §**Component** *C_Actuator***Port** *Measurement_Item* = *T_Actuate_In***Port** *Physical_Actuator* = *T_Signal_Out***Computation** = *Measurement_Item.actuate*→ *Physical_Actuator.start* → **Computation***_Measurement_Item.close* → *Physical_Actuator.close* → §

Connectors Specification

Measurement Item →→ Measurement Value

Connector *CN_ToMV* (*numMvs*: 1..*n*)**Role** *MI* = *T_Calculation_Out***Role** *MV_{1..numMvs}* = *T_Calculation_In***Glue** = *MI.perform_calculation*→ $\forall i:1..numMvs _ \overline{MV_i.perform_calculation}$ → **Glue***MI.calibrate* → $\forall i:1..numMvs _ \overline{MV_i.calibrate}$ → **Glue***_MI.compare* → $\forall i:1..numMvs _ \overline{MV_i.calibrate}$ → $(\forall i:1..numMvs _ \overline{MV_i.acceptable}$ → **Glue** $\exists i:1..numMvs _ \overline{MV_i.non_acceptable}$ → **Glue**)*MI.close* → $\forall i:1..numMvs _ \overline{MV_i.close}$ → §

Measurement Item →→ Actuator

Connector *CN_ToActuator* (*numAct*: 1..*n*)**Role** *MI* = *T_Actuate_Out***Role** *Actuator_{1..numAct}* = *T_Actuate_In***Glue** = *MI.actuate* → $\forall i:1..numAct _ \overline{Actuator_i.actuate}$ → **Glue***MI.close* → $\forall i:1..numAct _ \overline{Actuator_i.close}$ → §

Actuator →→ Physical Actuator

Connector *CN_Signallers***Role** *Actuator* = *T_Signal_Out***Role** *PActuator* = *T_Signal_In***Glue** = *Actuator.start* → $\overline{PActuator.receive}$ → **Glue***_Actuator.close* → $\overline{PActuator.close}$ → §

Observation: As we explained before, it is not possible to store values, so when the measurement value receives the event CALIBRATE, just accepts it and does nothing.

SECOND LEVEL

MOTIVATION

As we saw so far, we found different problems in the description. The most remarkable was the lacking of representation of internal state of the component (observations made in steps 2, 4, 5 and 6). In fact, Wright ADL [All97] is focused on the interaction behaviour and there is no possibility to use the operations given by the components. All their executions are hidden behind the non-deterministic choices in the specification. Thus, it is no possible to have any specification about the functional aspects of the components [San97]

In this level, we decided to solve the previous problem through the use of B Abstract Machines [San97] and refine the architecture using the concept of hierarchy in Wright ADL [All97] in the following details:

- The sensor updates itself with the data in the physical sensor and the update can vary , depending on the sensor type and the application. Thus, in this framework the updating behaviour was abstracted using the Strategy design pattern [GHJV95]
 - Client Update: The sensor never updates itself , until it is called by a client
 - Periodic Update: The sensor will update itself with fresh data from the physical sensor after a given time interval
 - On-change Update: When the value is changed, the sensor is notified
- The data read from the hardware sensor has to be converted into a value that has some meaning in the context of the software system. This conversion process can be different, depending on the application an the way the sensor is used. Therefore the conversion has also been abstracted as a Calculation Strategy. But the Calculation Strategy is used also by other objects in the framework.
- After the Measurement Item is started, it contains two execution phases: to collect data on the physical item it represents and to make the actuators actuate appropriately, based on the measured data and the comparison to the set data. In fact, these two processes are encapsulated in two strategies: Calculation Strategy (described previously) and Actuation Strategy (uses the set of actuator references stored by the measurement item to activate the various actuators)
- When the Item Factory is in calibration state, this fact influences the Actuation Strategy of the measurement item rather than the calculation strategy. Therefore, the normal actuation strategy is replaced with a calibration strategy that performs the actions required during calibration.

ANALYSIS

After an analysis, we detect that the components to be refined are: C_Item Factory, C_Measurement_Item, C_Measurement_Value and C_Sensor. All of them have in common the problem of not being able to store some special state in their description and, except the first one, have in common that they have associated different strategies.

Here, we present the changes made analyzing each component separately. In this section we will not show the evolution of each change as in the previous section, we will just present the final component and the changes made.

Component: ITEM FACTORY

In the step 2 in the previous section, we explained that the item factory decides if the system is in calibration or not and we expressed this fact through a non deterministic choice between the events CALIBRATE and NORMAL. In fact, the component has a boolean variable to express the actual state of the system. Using the concept presented in [San97], we build a new component AM_Item_Factor (which is the representation of the B-Abstract Machine of Factory and exports the operation *Is_InCalibration*), then we build the connector and we avoid the non-deterministic choice in Item_Factory. As this process only is taken into account in the factory environment, we encapsulated everything using hierarchical description.

Component CH_Item_Factory**Port** Trigger = T_Trigger_In**Port** Measurement_Item = T_CalibrationState_Out**Computation****Configuration** Factory**Component** C_Item_Factory**Port** Trigger = T_Trigger_In**Port** Measurement_Item = T_CalibrationState_Out**Port** Component = $\overline{\text{in_Calibration}}$ \rightarrow (calibration \rightarrow Component
 $\overline{\text{no_calibration}}$ \rightarrow Component)**Computation** = Trigger.trigger \rightarrow $\overline{\text{Component.in_Calibration}}$ \rightarrow
(Component.calibration \rightarrow $\overline{\text{Measurement_Item.calibrate}}$
Component.no_calibration
 \rightarrow $\overline{\text{Measurement_Item.normal}}$)
 \rightarrow $\overline{\text{Measurement_Item.start}}$ \rightarrow **Computation**
 $\overline{\text{_Trigger.close}}$ \rightarrow $\overline{\text{Measurement_Item.close}}$ \rightarrow §**Component AM_Item_Factory****Port** B = Is_InCalibration \rightarrow B**Connector B_Item_Factory****Role** Component = $\overline{\text{in_Calibration}}$ \rightarrow (calibration \rightarrow Component
no_calibration \rightarrow Component)**Role** B = Is_InCalibration \rightarrow B**Glue** = Component.in_calibration \rightarrow
(Component.Calibration \rightarrow **Glue**
 $\overline{\text{Component.no_Calibration}}$ \rightarrow **Glue**)

Informally:

If B.Is_InCalibration **then** $\overline{\text{Component.Calibration}}$
else $\overline{\text{Component.no_Calibration}}$ **Instances**

I_Factory : C_ItemFactory

Factory_State : AM_Item_Factory

Connection : B_Item_Factory

Attachments

I_Factory.Component as Connection.Component

Factory_State.B as Connection.B

End Factory**Bindings**

I_Factory.Trigger = Trigger

I_Factory.Measurement_Item = Measurement_Item

End Bindings**Component: MEASUREMENT ITEM**

As we said previously, the starting process of the Measurement Item is described in two phases: calculation and actuation phases. If we refer to the *start* method in the implementation, we will see:

```

start
  (self calculationStrategy) performCalculation
  (self actuationStrategy) actuate
  ^self

```

In fact, the calculation strategy just forwards the calculation process to the measurement item itself, and this last one forwards the *performCalculation* operation to its measurement values. In case of actuation strategy, if the system is in normal state, the actuators must work; but if it is in calibration state, just forwards the calibration process to the measurement item itself, and this last one forwards the *calibrate* operation to its measurement values.

Thus, we have to add two new components to represent the Actuation and Calibration strategies and their respective connectors. As these concepts are used by other components, we decide to build **mini-styles** with the strategies. More details will be given later, we just only add the expression **Style CalculationStrategy, ActuationStrategy**

As this process only is taken into account in the item environment, we encapsulated everything using hierarchical description

Component *CH_Measurement_Item*

Port *Item_Factory* = *T_CalibrationState_In*

Port *MValues* = *T_Calculation_Out*

Port *Actuators* = *T_Actuate_Out*

Computation

Configuration *Item_To_Measure*

Style *CalculationStrategy, ActuationStrategy*

Component *C_Measurement_Item*

Port *Item_Factory* = *T_CalibrationState_In*

Port *MValues* = *T_Calculation_Out*

Port *Actuators* = *T_Actuate_Out*

Port *ActuationStrategy* = *T_ActStrategy_Out*

Port *CalculationStrategy* = *T_CalcStrategy_Out*

Computation = *ItemFactory.calibrate* → *Item_Factory.start*

→ **Calculate** → **Calibrate**

→ *MValues.calibrate* → **Computation**

_Item.Factory.normal → *Item_Factory.start*

→ **Calculate** → **Compare** → **Computation**

_In.close → *MValues.close* → *Actuators.close* → §

where **Calculate** = $\overline{\text{CalculationStrategy.perform_calculation}}$ →

CalculationStrategy.context →

$\overline{\text{MValues.perform_calculation}}$

Compare = $\overline{\text{MValues.compare}}$ →

MValues.acceptable → *MValues.non_acceptable*

→ **Actuate** → *Actuators.actuate*

Actuate = $\overline{\text{ActuationStrategy.actuate}}$

→ *ActuationStrategy.actuators*

Calibrate = $\overline{\text{ActuationStrategy.calibrate}}$
 $\rightarrow \text{Actuation.Strategy.calibrate}$

Instances

M_Item : *C_Measurement_Item*
CalcStrategy : *C_CalculationStrategy*
ActStrategy : *C_ActuationStrategy*
CnToCalc : *Cn_MIToCalcStrategy*
CnToAct : *Cn_ToActStrategy*

Attachments

M_Item.CalculationStrategy as CnToCalc.In
CalcStrategy.InOut as CnToCalc.Out
M_Item.ActuationStrategy as CnToAct.In
ActStrategy.InOut as CnToAct.Out

End Item_To_Measure

Bindings

M_Item.Item_Factory = Item_Factory
M_Item.MValues = MValues
M_Item.Actuators = Actuators

End Bindings

Component: MEASUREMENT VALUE

In this component, we have a combination of the problems presented in the two previously components. Firstly, the component has to decide if the physical item is acceptable or not. In the first level, this fact was expressed through a non-deterministic choice, but in fact this situation is similar to one presented in Item Factory. The measurement values keep information about the last measured value and the ideal value for an aspect of the item, and can determine the condition of the acceptability of the item comparing this value. Thus, we apply the technique of [San97], and develop a component AM_MeasurementValue which represents the B Abstract Machine, and this component can export three operations: *set_value* (which stores the measured value), *ideal_value* (which calibrates the value) and *is-acceptable* (which returns True (or False) if the item is (not) acceptable).

Besides this, the measurement value has a calculation strategy. When the measurement item asks the measured value for a specific aspect, the measurement value just calls the calculation strategy, and this last one will say who will take the measure, in this case, the associated sensor.

As this process only is taken into account in the item environment, we encapsulated everything using hierarchical description

Component CH_MeasurementValue

Port *Measurement_Item* = *T_Calculation_In*
Port *Sensors* = *T_Measure_Out*

Computation

Configuration Value

Component *C_MeasurementValue*

Style *CalculationStrategy*

Port *Measurement_Item* = *T_Calculation_In*

Port *Sensors* = *T_Measure_Out*

Port *CalculationStrategy* = *T_CalcStrategy_In*

Port *Component* = $\overline{\text{set_value!val}}$ \rightarrow *Component*

$$\begin{aligned}
& \overline{_ideal_value} \rightarrow \text{Component} \\
& \overline{_compare} \rightarrow (\text{acceptable} \rightarrow \text{Component} \\
& \quad \text{non_acceptable} \rightarrow \text{Component}) \\
\text{Computation} = & \text{Measurement_Item.perform_calculation} \\
& \rightarrow \text{CalculationStrategy.perform_calculation} \\
& \rightarrow \text{CalculationStrategy.sensor} \\
& \rightarrow \text{Sensors.get_value} \rightarrow \text{Sensors.receive?val} \\
& \rightarrow \text{Component.set_value!val} \rightarrow \text{Computation} \\
& \text{Measurement_Item.calibrate} \\
& \rightarrow \text{Component.ideal_value} \rightarrow \text{Computation} \\
& \overline{_Measurement_Item.compare} \rightarrow \overline{\text{Component.compare}} \rightarrow \\
& \quad (\text{Component.acceptable} \rightarrow \\
& \quad \overline{\text{Measurement_Item.acceptable}} \rightarrow \text{Computation}) \\
& \overline{_Component.non_acceptable} \rightarrow \\
& \quad \overline{\text{Measurement_Item.non_acceptable}} \\
& \quad \rightarrow \text{Computation} \\
& \overline{_Measurement_Item.close} \rightarrow \overline{\text{Sensors.close}} \\
& \quad \rightarrow \overline{\text{CalculationStrategy.close}} \rightarrow \$
\end{aligned}$$

Component *AM_MeasurementValue*

$$\begin{aligned}
\text{Port } B = & \text{set_value?val} \rightarrow B \quad \overline{_ideal_value} \rightarrow B \\
& \overline{_Is_Acceptable} \rightarrow B
\end{aligned}$$

Connector *B_MValue*

$$\begin{aligned}
\text{Role Component} = & \overline{\text{set_value!val}} \rightarrow \text{Component} \\
& \overline{_ideal_value} \rightarrow \text{Component} \\
& \overline{_compare} \rightarrow (\text{acceptable} \rightarrow \text{Component} \\
& \quad \text{non_acceptable} \rightarrow \text{Component}) \\
\text{Role } B = & \text{set_value?val} \rightarrow B \quad \overline{_ideal_value} \rightarrow B \quad \overline{_compare} \rightarrow B \\
& \overline{_acceptable} \rightarrow B \quad \overline{_non_acceptable} \rightarrow B \\
\text{Glue} = & \text{Component.set_value?val} \rightarrow \overline{B.set_value!val} \rightarrow \text{Glue} \\
& \text{Component.ideal_value} \rightarrow \overline{B.ideal_value} \rightarrow \text{Glue} \\
& \text{Component.compare} \rightarrow \\
& \quad (\overline{\text{Component.acceptable}} \rightarrow \text{Glue} \\
& \quad \overline{\text{Component.non_acceptable}} \rightarrow \text{Glue})
\end{aligned}$$

Informally :

$$\begin{aligned}
& \text{if } B.\overline{_Is_Acceptable} \text{ then } \overline{\text{Component.acceptable}} \rightarrow \text{Glue} \\
& \quad \text{else } \overline{\text{Component.non_acceptable}} \rightarrow \text{Glue}
\end{aligned}$$

Instances

$$\begin{aligned}
M_Value & : C_Measurement_Value \\
State & : AM_Measurement_Value \\
CalcStrategy & : C_CalculationStrategy \\
Connection & : B_MValue \\
MVToCalc & : CN_MIToCalcStrategy
\end{aligned}$$

Attachments

*M_Value.CalculationStrategy as MVToCalc.In
CalcStrategy.InOut as MVToCalc.Out
M_Value.Component as Connection.Component
State.B as Connection.B*

End Value**Bindings**

*M_Value.Measurement_Item = Measurement_Item
M_Value.Sensors = Sensors*

End Bindings**Component: SENSOR**

This component present the same situation as Measurement Value. It has associated a Calculation Strategy, an Update Strategy and must keep information about internal state. The Calculation Strategy just forwards the request to the sensor itself and the sensor ask the physical sensor to measure the item. The Update Strategy decides when the sensor must update its value . The internal state is represented using [San97] and the component AM_Sensor exports two operations: set_value (which stores the measured value) and measured_value (which returns the value).

As this process only is taken into account in the item environment, we encapsulated everything using hierarchical description

Component CH_Sensor

Port *Measurement_Item = T_Measure_In*

Port *Physical_Sensor = T_Output_Input*

Computation

Configuration *SensorDevice*

Style *CalculationStrategy, UpdateStrategy*

Component C_Sensor

Port *Measurement_Item = T_Measure_In*

Port *Physical_Sensor = T_Output_Input*

Port *CalculationStrategy = T_CalcStrategy_Out*

Port *UpdateStrategy = T_Update_Out*

Port *Component = measured_value → measured_value?val → Component
_set_value!val → Component*

Computation = *Measurement_Item.get_value*
→ *UpdateStrategy.client_update*
→ (*UpdateStrategy.context*
→ *Component.measured_value* →
Component.measured_value?val →
Measurement_Item.send!val → **Computation**
UpdateStrategy.calculate_value →
CalculationStrategy.perform_calculation
→ *CalculationStrategy.context*
→ *Physical_Sensor.ask*
→ *Physical_Sensor.receive?result*
→ *Component.set_value!result* →
Measurement_Item.send!result
→ *Computation*)

$$\begin{aligned}
 & \text{UpdateStrategy.calculate_value} \rightarrow \\
 & \quad \overline{\text{CalculationStrategy.perform_calculation}} \\
 & \quad \rightarrow \text{CalculationStrategy.context} \\
 & \quad \rightarrow \overline{\text{Physical_Sensor.ask}} \\
 & \quad \rightarrow \text{Physical_Sensor.receive?result} \\
 & \quad \rightarrow \overline{\text{Component.set_value!result}} \rightarrow \text{Computation} \\
 & \quad _ \text{Measurement.close} \rightarrow \text{Physical_Sensor.close} \rightarrow \$
 \end{aligned}$$
Component *AM_Sensor*

Port *B* = *set_value?val* → *B_measured_value!val* → *B*

Connector *B_Sensor*

Role *Component* = $\overline{\text{set_value!val}} \rightarrow \text{Component}$
 $_ \text{measured_value} \rightarrow \text{measured_value?val}$
 $\rightarrow \text{Component}$

Role *B* = *set_value?val* → *B_measured_value!val* → *B*

Glue = $\text{Component.set_value?val} \rightarrow \overline{\text{B.set_value!val}} \rightarrow \text{Glue}$
 $\text{Component.measured_value}$

$\rightarrow \overline{\text{B.measured_value?val}} \rightarrow$
 $\overline{\text{Component.measured_value!val}} \rightarrow \text{Glue}$

Instances

Sensor : *C_Sensor*
CalcStrategy : *C_CalculationStrategy*
UpdStrategy : *C_UpdateStrategy*
State : *AM_Sensor*
SensorToCS : *CN_MIToCalcStrategy*
SensorToUS : *CN_SensorToUpdStrategy*
Connection : *B_Sensor*

Attachments

Sensor.CalculationStrategy as *SensorToCS.In*
CalcStrategy.InOut as *SensorToCs.Out*
Sensor.UpdateStrategy as *SensorToUS.In*
UpdStrategy.InOut as *SensorToUS.Out*
Sensor.Component as *Connection.Component*
State.B as *Connection.B*

End *SensorDevice*

Bindings

Sensor.Measurement_Item = *Measurement_Item*
Sensor.Physical_Sensor = *Physical_Sensor*

End Bindings

STRATEGIES DEFINED AS STYLES

In the last section, we observed that most of the components used the Calculation, Update and Actuation strategies. Thus, we think that it was useful to define an Style with each of them.

Each style is composed of the component Strategy and the respective connector. They are simple styles and they do not have any specific constraints.

Style ActuationStyle

Interface Type $T_ActStrategy_Out = \overline{\text{actuate}} \rightarrow \text{actuators} \rightarrow T_ActStrategy_Out$
 $\text{calibrate} \rightarrow \text{calibrate} \rightarrow T_ActStrategy_Out$
 $\text{close} \rightarrow \S$

Interface Type $T_ActStrategy_In = \overline{\text{actuate}} \rightarrow \text{actuators} \rightarrow T_ActStrategy_In$
 $\text{calibrate} \rightarrow \text{calibrate} \rightarrow T_ActStrategy_In$
 $\text{close} \rightarrow \S$

Connector $CN_ToActStrategy$

Role In = $T_ActStrategy_Out$

Role Out = $T_ActStrategy_In$

Glue = $In.\overline{\text{actuate}} \rightarrow Out.\overline{\text{actuate}} \rightarrow Out.\text{actuators} \rightarrow In.\overline{\text{actuators}} \rightarrow \mathbf{Glue}$
 $In.\overline{\text{calibrate}} \rightarrow Out.\overline{\text{calibrate}} \rightarrow Out.\text{calibrate} \rightarrow In.\overline{\text{calibrate}} \rightarrow \mathbf{Glue}$
 $In.\text{close} \rightarrow Out.\text{close} \rightarrow \S$

Component $C_ActuationStrategy$

Port InOut = $T_ActStrategy_In$

Computation = $InOut.\overline{\text{actuate}} \rightarrow InOut.\overline{\text{actuators}} \rightarrow \mathbf{Computation}$

$InOut.\text{calibrate} \rightarrow InOut.\text{calibrate} \rightarrow \mathbf{Computation}$

End Style

Style CalculationStrategy

Interface Type $T_CalcStrategy_Out = \overline{\text{perform_calculation}} \rightarrow \text{context} \rightarrow \text{CalculationStrategy}$
 $\text{close} \rightarrow \S$

Interface Type $T_CalcStrategy_In = \overline{\text{perform_calculation}} \rightarrow \text{context} \rightarrow \text{CalculationStrategy}$
 $\text{close} \rightarrow \S$

Connector $CN_ToCalcStrategy$

Role In = $T_CalcStrategy_Out$

Role Out = $T_CalcStrategy_In$

Glue = $In.\overline{\text{perform_calculation}} \rightarrow Out.\overline{\text{perform_calculation}}$
 $\rightarrow Out.\text{context} \rightarrow In.\overline{\text{context}} \rightarrow \mathbf{Glue}$
 $In.\text{close} \rightarrow Out.\text{close} \rightarrow \S$

Component $C_CalculationStrategy$

Port InOut = $T_CalcStrategy_In$

Computation = $InOut.\overline{\text{perform_calculation}} \rightarrow InOut.\overline{\text{context}} \rightarrow \mathbf{Computation}$

End Style

Style UpdateStrategy

Interface Type $T_Update_Out = \overline{\text{client_update}} \rightarrow T_Update_Out$

$$\begin{aligned} & \text{_calculate_value} \rightarrow T_Update_Out \\ & \text{_context} \rightarrow T_Update_Out \\ \mathbf{Interface\ Type\ } T_Update_In & = \text{client_update} \rightarrow T_Update_In \\ & \text{_calculate_Value} \rightarrow T_Update_In \\ & \text{_context} \rightarrow T_Update_In \end{aligned}$$

$$\begin{aligned} \mathbf{Connector\ } CN_ToUpdateStrategy & \\ \mathbf{Role\ In} & = T_Update_Out \\ \mathbf{Role\ Out} & = T_Update_In \\ \mathbf{Glue} & = In.client_update \rightarrow Out.client_update \rightarrow \mathbf{Glue} \\ & \text{_Out.calculate_value} \rightarrow In.calculate_value \rightarrow \mathbf{Glue} \\ & \text{_Out.context} \rightarrow In.context \rightarrow \mathbf{Glue} \end{aligned}$$

$$\begin{aligned} \mathbf{Component\ } C_UpdateStrategy & \\ \mathbf{Port\ InOut} & = \text{client_update} \rightarrow InOut \\ & \text{_calculate_value} \rightarrow InOut \\ & \text{_context} \rightarrow InOut \\ \mathbf{Computation} & = InOut.client_update \rightarrow (InOut.calculate_value \rightarrow \mathbf{Computation} \\ & \text{_InOut.context} \rightarrow \mathbf{Computation}) \\ & \text{_InOut.calculate_value} \rightarrow \mathbf{Computation} \end{aligned}$$

End Style

Interface Types

$$\begin{aligned} \mathbf{Interface\ Type\ } T_Signal_Out & = \text{start} \rightarrow T_Signal_Sender \text{_close} \rightarrow \S \\ \mathbf{Interface\ Type\ } T_Signal_In & = \text{receive} \rightarrow T_Signal_Receiver \text{_close} \rightarrow \S \\ \mathbf{Interface\ Type\ } T_Trigger_Out & = \text{trigger} \rightarrow T_Trigger_Out \text{_close} \rightarrow \S \\ \mathbf{Interface\ Type\ } T_Trigger_In & = \text{trigger} \rightarrow T_Trigger_In \text{_close} \rightarrow \S \\ \mathbf{Interface\ Type\ } T_Output_Input & = \text{ask} \rightarrow \text{receive?item} \rightarrow T_Output_Input \text{_close} \rightarrow \S \\ \mathbf{Interface\ Type\ } T_Input_Output & = \text{receive} \rightarrow \text{send!result} \rightarrow T_Input_Output \text{_close} \rightarrow \S \\ \mathbf{Interface\ Type\ } T_Calibration_Out & = \text{calibrate} \rightarrow \text{start} \rightarrow T_Calibration_Out \\ & \text{_normal} \rightarrow \text{start} \rightarrow T_Calibration_Out \text{_close} \rightarrow \S \\ \mathbf{Interface\ Type\ } T_Calibration_In & = \text{calibrate} \rightarrow \text{start} \rightarrow T_Calibration_In \\ & \text{_normal} \rightarrow \text{start} \rightarrow T_Calibration_In \text{_close} \rightarrow \S \\ \mathbf{Interface\ Type\ } T_Calculation_Out & = \text{perform_calculation} \rightarrow T_Calculation_Out \\ & \text{_calibrate} \rightarrow T_Calculation_Out \\ & \text{_compare} \rightarrow (\text{acceptable} \rightarrow T_Calculation_Out \\ & \text{_non_acceptable} \rightarrow T_Calculation_Out) \\ & \text{_close} \rightarrow \S \\ \mathbf{Interface\ Type\ } T_Calculation_In & = \text{perform_calculation} \rightarrow T_Calculation_In \\ & \text{calibrate} \rightarrow T_Calculation_In \\ & \text{compare} \rightarrow (\text{acceptable} \rightarrow T_Calculation_Out \\ & \text{_non_acceptable} \rightarrow T_Calculation_Out) \\ & \text{_close} \rightarrow \S \end{aligned}$$

Interface Type $T_Actuate_Out = \overline{\text{actuate}} \rightarrow T_Actuate_Out _ \overline{\text{close}} \rightarrow \S$

Interface Type $T_Actuate_In = \text{actuate} \rightarrow T_Actuate_In _ \text{close} \rightarrow \S$

Interface Type $T_Measure_Out = \overline{\text{get_value}} \rightarrow \text{receive?val} \rightarrow T_Measure_Out _ \overline{\text{close}} \rightarrow \S$

Interface Type $T_Measure_In = \text{get_value} \rightarrow \overline{\text{send!val}} \rightarrow T_Measure_In _ \text{close} \rightarrow \S$

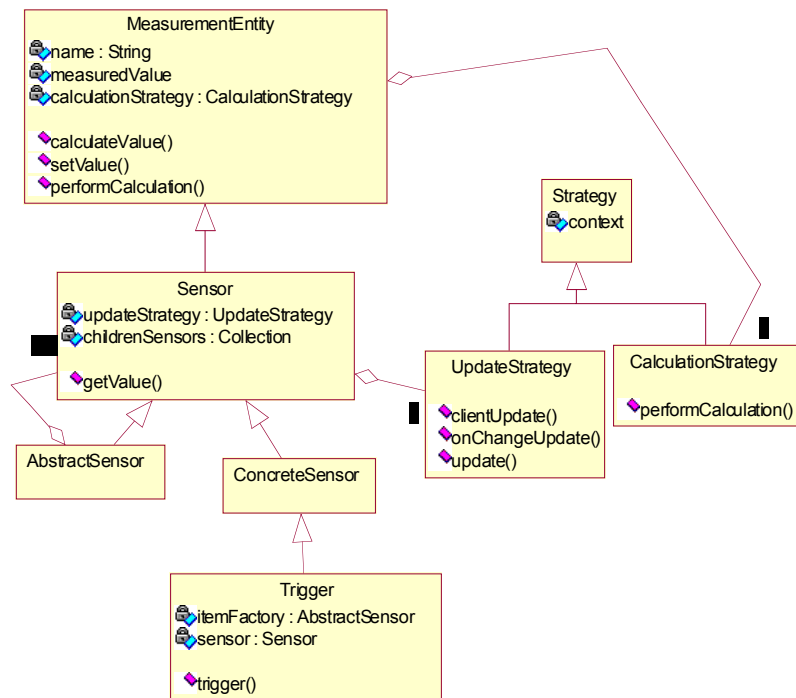


Figure 1: Class relations for Sensor

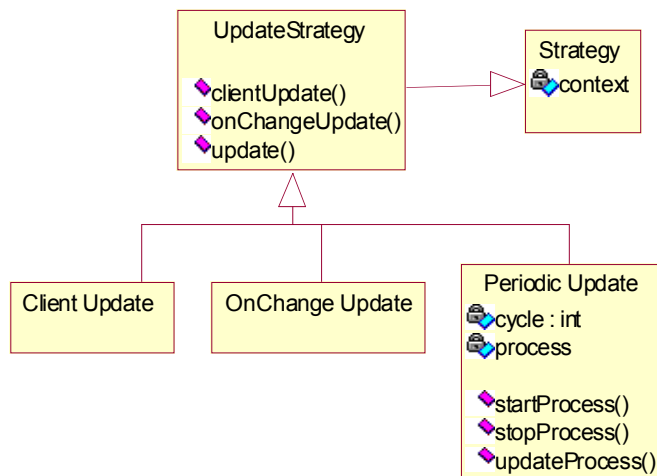


Figure 2: UpdateStrategy class hierarchy

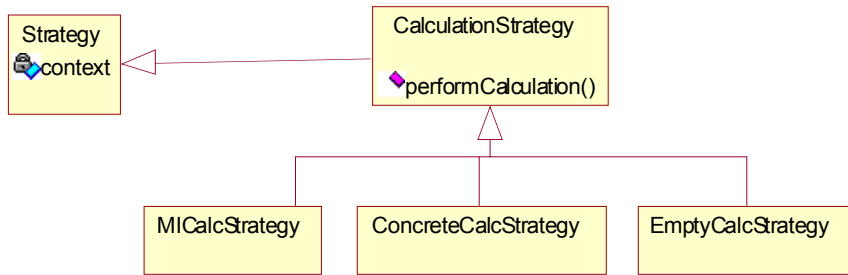


Figure 3: CalculationStrategy class hierarchy

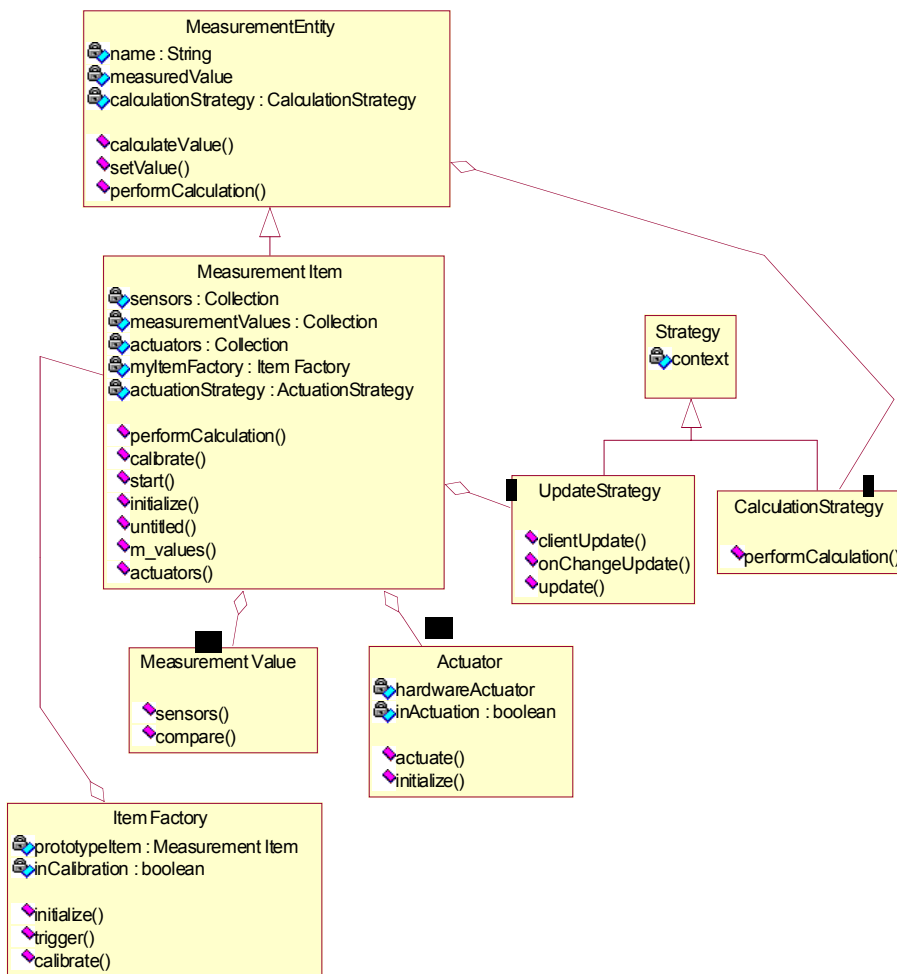


Figure 4: Relations for Class Measurements

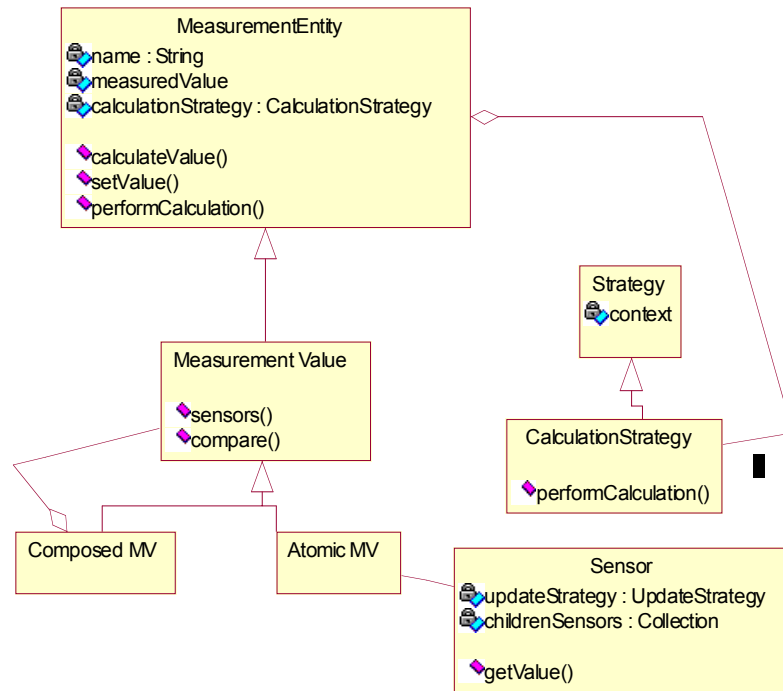


Figure 5: MeasurementValue class relations

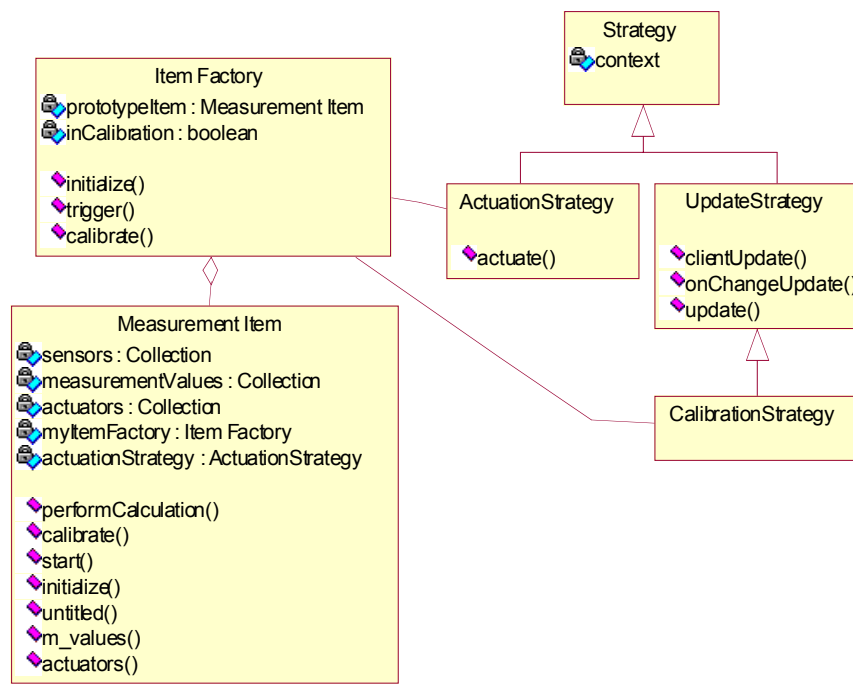


Figure 6: ItemFactory Class

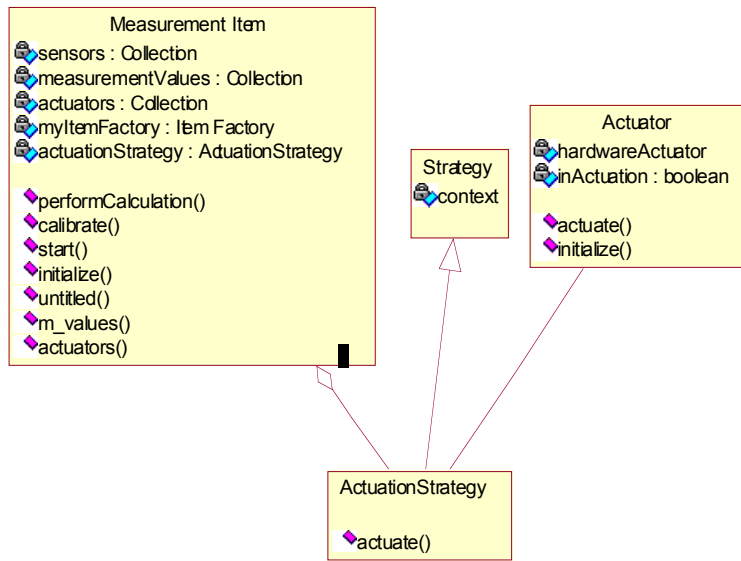


Figure 7: Context of class Actuator

INTERACTION DIAGRAMS

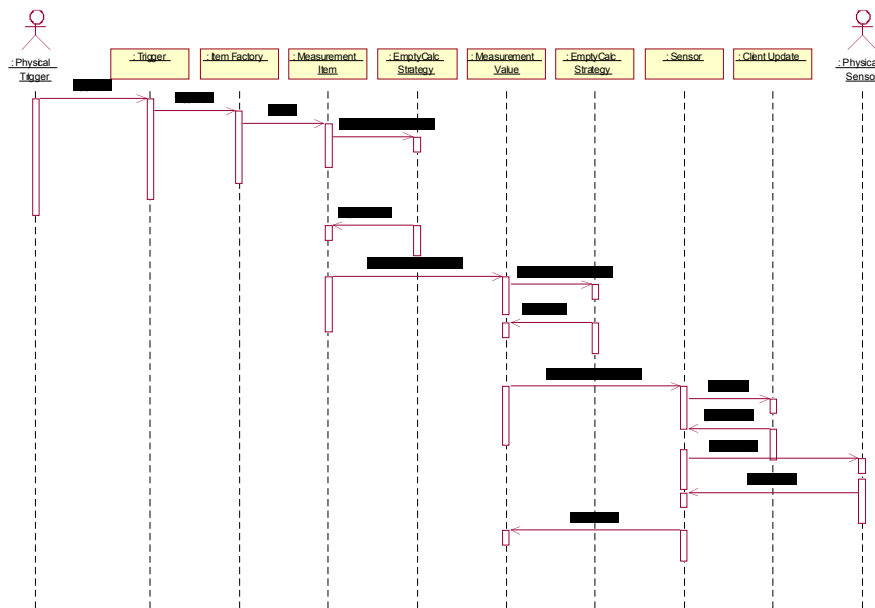


Figure 8: Normal Process with the Sensor data updated through the request of the System

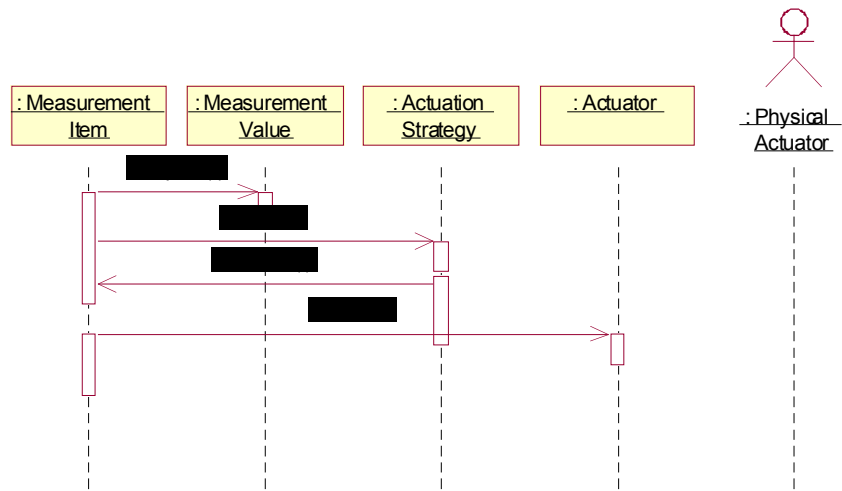


Figure 9: Actuation Phase in Normal Process of the System

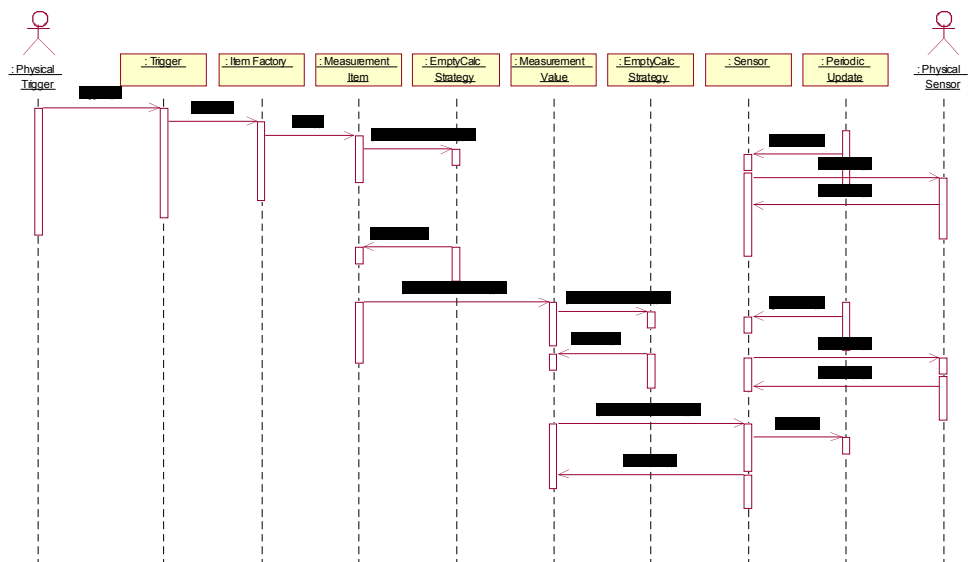


Figure 10: Normal Process with the Sensor data updated periodically

Appendix B

This appendix shows a complete analysis of the application of Wright ADL to describe a component of a framework. The objective is to see an step-by-step process of how to get an architectural description using this ADL.

A GENERIC COORDINATION ABSTRACTION FOR MANAGING SHARED RESOURCES

In [CTN97], a Coordination Component Framework for Open Distributed Systems was developed. They introduce different solutions to coordination problems, and each solution is embodied in a component in the framework. One of these components is used to provide solution to the management of shared resource.

In[CDK94], they define that there are some common elements which define the structure of these solutions : the definition of the resource, and the definition of the allocation policy. Some features of these elements are fundamental for the specification of the solutions. For the resource they are basically : the size of the resource, and the number of concurrent entities which can access the resource at the same time. For the allocation policy they are basically : the order in which the resource is going to be assigned, the maximum allowed to each entity, if there will be priorities on the requests of the allocations, and what to do with the allocation requests that cannot be processed simultaneously (to define a waiting queue of entities, to ignore them, etc.). There are also some fault-tolerance aspects like what to do in case of software failures (to ignore precedent request, to guarantee recovery, etc.) that have taken into account. All these aspects define the parameters of variability of the generic coordination abstraction that could be specified and that can be used to generate specific coordination solutions]

One example shown to see how the component is applied is a toy banking system. In this case the resource is an account database which is shared by multiple teller machines. The teller machines need to get information from the account database in order to check a client's account. They also need to update account information if they have given money to a client. To keep the database consistent we need an access policy to regulate the multiple requests.

As a solution for this regulation they introduce an access policy component. This solution not only provides access regulation, but also explicitness of architecture and flexibility.

8.4 REQUIREMENTS

The main goal of the design is to implement the access policy to the database in a reusable and configurable way. In terms of the component oriented approach, this means that we want to have a structure where we can plug in different policies, without having to change other parts of the solution. Three kinds of policies are distinguished:

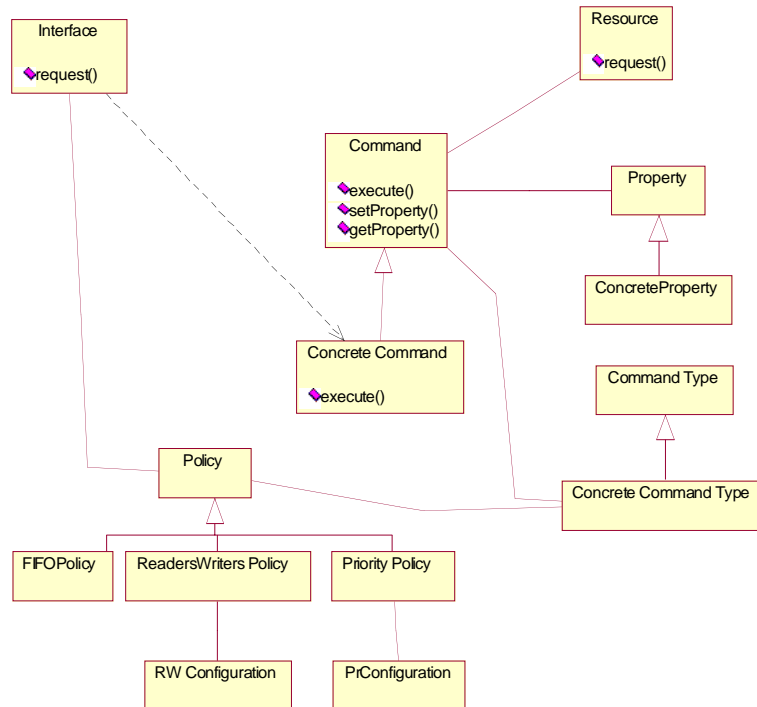
Policies that need no other information about the requests: This is the easiest kind of policy. They can do their job without any knowledge about the commands they have to dispatch. A typical example is a FIFO policy: no matter what commands come in, the policy just dispatches them in order they come in.

Policies that need type information about the requests: This kind of policy depends on the "nature" of the command: every command *type* has one or more properties which are needed by the policy. A typical example for this case is the readers/writers policy. The policy has to know if a command is a reader or a writer command. So this information must be made available in our solution.

Policies that need external information about the requests: This kind of policy depends on information which can be different for each instance of a command. It can be thought as a priority policy, where, depending on the sender of the request, a command has a certain priority. Again, somehow this information must be available to the policy.

8.5 SOLUTION

There is an Interface of the solution to the rest of the application. Clients have to call this Interface to access the access solution. The resource itself is also modelled and there is a part which represents the control policy. To give the policy the ability to buffer the commands, change their order or execute them in parallel, it is necessary an explicit representation of the commands. This is done using the Command Pattern[GHJV95].



Then let's see the relationship between Command and Policy. For the first kind of policy, the basic Command pattern suffices. For the second and third however, it is necessary some additions. For the second kind of policy, we need to make type information available at run-time in order to be able to use this information at run-time, for instance to link this information to certain policy-dependent properties. We make this information available by providing a **CommandType** class to every subclass of the abstract Command class.

For the third kind of policy we need to make information available which can differ for every instance of a command. We do this by connecting a **Property** class to every Command class and the addition of a `SetProperty` and a `GetProperty` method to the Command class. What the exact information is, that subclasses of this class represent, is totally dependent on the application in which it is used. It could be that the name of the sender of the request is made available. The information, that is made available, should not be information especially linked to a policy. As an example, we take again the priority policy. In the **Property** object, we should make the sender of a request available. The linking of this sender with a priority is done later at the policy. We do this to keep the Command as independent of the policy as possible: the information is really a property of the Command and this information can be used by different policies.

The decision to take a **Property** class to hold information about commands has some advantages and disadvantages. The advantage is that the Command does not have to be aware of information that is made available about itself. It is also explicit in the architecture which information is made available and it is applicable without having to change the Command class.

The disadvantage is the overhead: the Interface may create Property classes and initialize information that may never be used.

The Policy part is set up using the Strategy pattern [GHVJ95]. The hard part is again the fact that we have to deal with the application dependent information. The solution is to represent this information explicitly in so-called "configuration" objects. These configuration objects contain at run-time the information that is needed by the policies. So we may have, for example, a configuration object that contains of a link between command types and a property isReader. Another example is an object that links the names of possible request senders to a certain priority. We see here the difference between information that is made available in a Property object and that in a Configuration object. The former is general information, the latter contains the policy dependent information.

In figure X, we see the total design for the shared resource access policy solution. The class Interface is the interface to the resource for the rest of the system. For every command which is invoked by an incoming events, a Command object is created. These commands are then given to the policy which is connected (through parameterization) to the interface. This policy handles the commands, i.e. determines when and in which order the request can access the resource. If the command is allowed, it is executed.

8.6 Wright Description

From the figure 1, we can see four elements: Interface, Command, Policy and Resource. We will model as components Command and Resource and Interface and Policy as connectors. The reasons are that Command is just one unit with information which has to receive a signal to be executed, and Resource is one unit which process the information sent in the Commands objects. The case for Interface is our connection to the virtual user and is the entry point for the system and Policy is a unit of communication between the command and the resource because it put the order of execution in the Resource.

Note: we will present different possible descriptions (from simpler ones to more elaborated), but in all the cases we explain all the taken assumptions.

Connector Interface: This element represents the entry point to the component, so we need two events: one to indicate that we received a request to be executed (request) and one to indicate that this request has to be processed according to the policy. We will assume that we have only one type of request to be executed.

Connector Interface

Role *User* = request __ \$
Role *Command* = create _ *Command*
Computation = *User.request* _ *Command.create* _ **Computation**

Component Command: This element represents the request that will be executed in the resource. It has three events: create (indicates that it was created), put (is enqueued in the policy to wait for the moment to be executed) and execute (indicates that can be executed). The restriction is that once it is created, it must be enqueued in the policy queue and has to wait for its turn to execute. This restriction is shown in the Computation

Component Command

Port *In* = create _ *In*
Port *Out_Policy* = put _ *Out_Policy* _ execute _ ready _ \$
Port *Out_Resource* = execute _ *Out_Resource* _ free _ *Out_Resource*
error _ *Out-Resource* _ busy _ *Out_Resource*
Computation = *In.create* _ *Out_Policy.put* _ *Out_Policy.execute*

$$\begin{aligned}
 & \text{_Continue_} \overline{\text{Out_Policy.ready_}} \$ \\
 \text{where } \mathbf{Continue} &= \overline{\text{Out_Resource.free_}} \overline{\text{Out_Resource.execute_}} \$ \\
 & \text{_} \overline{\text{Out_Resource.execute_}} \text{_} \overline{\text{Out_Resource.error_}} \text{_} \mathbf{Continue} \\
 & \text{_} \overline{\text{Out_Resource.execute_}} \text{_} \overline{\text{Out_Resource.busy_}} \text{_} \mathbf{Continue}
 \end{aligned}$$

Component Resource: This element is any context can answer to three states associated to events : free(can accept any request), request(receives the request), busy(processing state of the request). The only condition to be fulfilled is that:

- while the resource is available sends the event free
- but when it receives the request, has to notify that the state is busy and that can not execute the request event.

Both cases are shown in the computation of the component.

Component Resource

$$\begin{aligned}
 \mathbf{Port\ In} &= \overline{\text{execute_In_}} \overline{\text{free_In_}} \overline{\text{busy_In_}} \$ \\
 \mathbf{Computation} &= \overline{\text{In.free_}} \text{In.execute_} \mathbf{Continue_} \mathbf{Computation} \\
 \text{where } \mathbf{Continue} &= \text{In.execute_} \overline{\text{In.busy_}} \mathbf{Continue_} \$
 \end{aligned}$$

Connector Policy: This element represents the algorithm that will administrate the order of execution of the commands. Now, we will consider that the policy is FIFO and that there are no constraints in the amount of commands that the policy queue can manage.

Connector Policy

$$\begin{aligned}
 \mathbf{Role\ Command} &= \overline{\text{put_}} \text{Command_} \overline{\text{execute_}} \overline{\text{ready_}} \$ \\
 \mathbf{Computation} &= \text{Command.put_} \overline{\text{Command.execute_}} \\
 & \text{_} \overline{\text{Command.ready_}} \mathbf{Computation_} \$
 \end{aligned}$$

Trick: We need a superfluous connector between Command and Resource components.

Connector Superfluous

$$\begin{aligned}
 \mathbf{Role\ Command} &= \overline{\text{execute_}} \text{Command_} \overline{\text{free_}} \text{Command_} \\
 & \text{_} \overline{\text{error_}} \text{Command_} \overline{\text{busy_}} \text{Command_} \\
 \mathbf{Role\ Resource} &= \overline{\text{execute_}} \text{Resource_} \overline{\text{free_}} \text{Resource_} \overline{\text{busy_}} \text{Resource_} \$ \\
 \mathbf{Glue} &= \overline{\text{Command.execute_}} \overline{\text{Resource.execute_}} \text{Glue} \\
 & \text{_} \overline{\text{Resource.free_}} \text{Command.free_} \text{Glue} \\
 & \text{_} \overline{\text{Resource.busy_}} \text{Command.busy_} \text{Glue} \\
 & \text{_} \$
 \end{aligned}$$

With these four architectural elements, we get a microarchitecture to represent the management of a shared resource with a FIFO policy and only one type of operation that can be executed atomically in the resource. In fact, the architectural description to define these components is very simple. It is interesting to add/change functionalities to extend the expressiveness of the different elements.

Some possible variants are:

- To have a set of operations to be executed over the resource, e.g. read and write

- To change the policy: Some policies need type information about the requests or need external information about the requests. We can consider also the possibility of having a waiting queue of requests.
- To check the number of concurrent entities which can access the resource at the same time.

There are other possible variants that can be taken into account, but our objective is just only to show how all of them can be applied in our description to get different functionalities.

8.6.1 First Variant: To have a set of operations, e.g. read and write

The fact that we have two operations with conflicts between them (an operation of writing can not be made during reading process, and reading process can not be taken into account when a writing process is happening) changes the definition of some protocols of Resource, Interface and Policy. In all the cases, we must identify what kind of operations is to know how to proceed.

Connector Interface

$$\begin{aligned} \mathbf{Role} \text{ User} &= \overline{\text{read}} _ \text{User} _ \overline{\text{write}} _ \text{User} _ \S \\ \mathbf{Role} \text{ CommandReader} &= \overline{\text{create}} _ \text{CommandReader} \\ \mathbf{Role} \text{ CommandWriter} &= \overline{\text{create}} _ \text{CommandWriter} \\ \mathbf{Computation} &= \text{User.read} _ \overline{\text{CommandReader.create}} _ \mathbf{Computation} \\ &\quad \text{User.write} _ \overline{\text{CommandWriter.create}} _ \mathbf{Computation} \end{aligned}$$

Connector Policy (nbr: 1.., nbw: 1..)

$$\begin{aligned} \mathbf{Role} \text{ CommandReader}_{1..nbr} &= \overline{\text{put}} _ \text{CommandReader} _ \overline{\text{execute}} _ \text{CommandReader} \\ &\quad _ \overline{\text{ready}} _ \text{CommandReader} _ \S \\ \mathbf{Role} \text{ CommandWriter}_{1..nbw} &= \overline{\text{put}} _ \text{CommandWriter} _ \overline{\text{execute}} _ \text{CommandWriter} \\ &\quad _ \overline{\text{ready}} _ \text{CommandWriter} _ \S \\ \mathbf{Computation} &= (\exists i..k < nbr \parallel \overline{\text{CommandReader.put}} _ \overline{\text{CommandReader.execute}}) \\ &\quad _ \forall i..k \overline{\text{CommandReader.ready}} _ \mathbf{Computation} \\ &\quad _ \overline{\text{CommandWriter.put}} _ \overline{\text{CommandWriter.execute}} \\ &\quad _ \overline{\text{CommandWriter.ready}} _ \mathbf{Computation} \\ &\quad \S \end{aligned}$$

The Component Resource must not pay attention to inform that the operation has finished, because in this case it can accept a set of operations of the same type (e.g. readers at the same time, or just only one writer).

Component Resource (nbr: 1.., nbw: 1..)

$$\begin{aligned} \mathbf{Port} \text{ In} &= \overline{\text{execute}} _ \text{In} _ \S \\ \mathbf{Computation} &= E i:1..k \parallel \text{In.execute} _ \mathbf{Computation} \end{aligned}$$

8.6.2 Second Variant: Generalize the set of commands.

Some requests in a real domain can be categorized as readers or writers of the resource, without be called explicitly read or write. The solution made in [CTN97] is to have an object called Configuration which associates the information about the command with its properties (in this case, the property is indicated as isReader or isWriter).

Component Configuration

$$\mathbf{Port} \text{ In} = \text{property?}i _ (\overline{\text{isReader}} _ \text{In} _ \overline{\text{isWriter}} _ \text{In})$$

$$\begin{aligned} \mathbf{Computation} &= \overline{In.property?i} _ \\ &\quad (\overline{In.isReader} _ \overline{Computation} _ \overline{In.isWriter} _ \overline{Computation}) \end{aligned}$$

The connector Policy is responsible for managing how the commands will be controlled to access the shared resource.

Connector Policy (*nbc: 1..*)

$$\begin{aligned} \mathbf{Role} \mathbf{Command}_{1..nbc} &= \overline{put} _ \overline{Command} _ \overline{execute} _ \overline{Command} \\ &\quad _ \overline{ready} _ \overline{CommandReader} _ \$ \\ \mathbf{Role} \mathbf{Configuration} &= \overline{property?i} _ \\ &\quad (\overline{isReader} _ \overline{Computation} _ \overline{isWriter} _ \overline{Computation}) \\ \mathbf{Computation} &= \overline{Command_i.put} _ \overline{Configuration.property!i} \\ &\quad _ (\overline{Configuration.isWriter} _ \overline{Command_i.execute} \\ &\quad _ \overline{Command_i.ready} _ \overline{Computation} \\ &\quad _ \overline{Configuration.isReader} _ \mathbf{Read} _ \mathbf{Computation}) \\ \text{Where } \mathbf{Read} &= ((\overline{Command_i.execute} _ _ (\exists 2..k < nbc _ _ \overline{Command_i.put} \\ &\quad _ \overline{Command_i.execute}) _ \forall i..k _ \overline{Command_i.ready} \end{aligned}$$

8.6.3 Third Variant: Use of commands with properties

This variant is very similar one to second variant. Some commands can occur concurrently in the resource and others ones should be executed alone. It is obvious that the architectural elements to be modified are Configuration and Policy.

Component Configuration

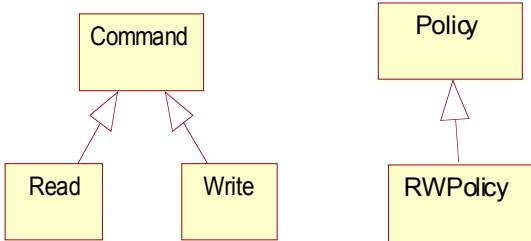
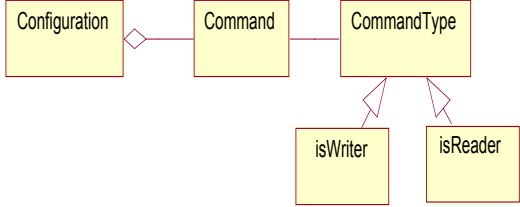
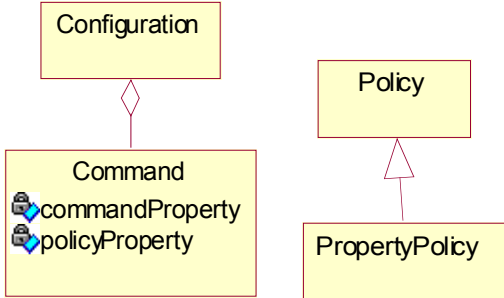
$$\begin{aligned} \mathbf{Port} \mathbf{In} &= \overline{property?i} _ (_ \overline{hasProperty!x} _ \overline{In}) \\ \mathbf{Computation} &= \overline{In.property?i} _ \overline{In.hasProperty!x} \end{aligned}$$

Connector Policy (*nbc: 1..*)

$$\begin{aligned} \mathbf{Role} \mathbf{Command}_{1..nbc} &= \overline{put} _ \overline{Command} _ \overline{execute} _ \overline{Command} \\ &\quad _ \overline{ready} _ \overline{CommandReader} _ \$ \\ \mathbf{Role} \mathbf{Configuration} &= \overline{property?i} _ (_ \overline{hasProperty!x} _ \overline{Configuration}) \\ \mathbf{Computation} &= \overline{Command_i.put} _ \overline{Configuration.property!x} _ \\ &\quad \exists j : 1..k _ \overline{Configuration.property!j} _ \overline{Command_j.execute} \\ &\quad _ \overline{Command_j.ready} _ \mathbf{Computation} \\ &\quad \exists j : 1..n _ \overline{Configuration.property!j} _ \mathbf{Read} _ \mathbf{Computation} \\ \text{where } \mathbf{Read} &= ((\overline{Command_j.execute} _ _ (\exists n:2..k < nbc _ _ \overline{Command_n.put} \\ &\quad _ \overline{Command_n.execute}) _ \forall i..k _ \overline{Command_n.ready} \end{aligned}$$

The index i indicate all the commands that have to be executed alone and index j indicate all the commands that can be executed concurrently.

Let's see in a progressive way what kinds of change we have made thinking in terms of classes diagrams:

Steps	Classes and Relationships	Wright Description
Initial Phase	Identify 'abstract' classes	Classify the classes as components and connectors in terms of their function inside the system.
Second Phase	Identify responsibilities, behaviours and collaborations	Model the ports, computation, roles and glue in the components and connectors
Third Phase: First Variant	<p>Specialize Command Class</p>  <pre> classDiagram class Command class Read class Write class Policy class RWPolicy Command < -- Read Command < -- Write Policy < -- RWPolicy </pre>	<p>Modify the roles of Connector Interface and the Computation</p> <p>Modify the Computation of the Connector Policy</p>
Third Phase: Second Variant	<p>Generalize the Command Class</p>  <pre> classDiagram class Configuration class Command class CommandType class isWriter class isReader Configuration o-- Command Command -- CommandType CommandType < -- isWriter CommandType < -- isReader </pre>	<p>Add the Component Configuration</p> <p>Modify the connector Policy in the computation</p>
Third Phase: Third Variant	<p>Generalize the properties of the Commands</p>  <pre> classDiagram class Configuration class Command class Policy class PropertyPolicy Configuration o-- Command Policy < -- PropertyPolicy class Command { commandProperty policyProperty } </pre>	<p>Modify the Role of the Component Configuration</p> <p>Modify the connector Policy in the computation</p>

The steps show that in fact most changes is focused on the behaviour of policy. This class was modelled with the Strategy pattern, and all the changes represented to have a new subclass in the model.