

**THE UNIVERSITY OF TULSA
THE GRADUATE SCHOOL**

**ARCHITECTURE INTEGRATION ELEMENTS:
CONNECTOR MODELS THAT FORM MIDDLEWARE**

by

Reshma Madhusudan Keshav

**A thesis submitted in partial fulfillment of
the requirements for the degree of Master of Science
in the Discipline of Computer Science**

The Graduate School

The University of Tulsa

1999

**THE UNIVERSITY OF TULSA
GRADUATE SCHOOL**

**ARCHITECTURE INTEGRATION ELEMENTS:
CONNECTOR MODELS THAT FORM MIDDLEWARE**

by

Reshma Madhusudan Keshav

A THESIS

**APPROVED FOR THE DISCIPLINE OF
COMPUTER SCIENCE**

By Thesis Committee

_____, Chairperson

ABSTRACT

Keshav, Reshma Madhusudan (Master of Computer Science)

Architecture Integration Elements: Connector Models that form Middleware (Chapter I – VI, pp 1 – 76)

Directed by Dr. Rose Gamble

(94)

Companies have turned to component based software engineering as a means to cut the cost of developing a system from the ground up. Hindering this method of development is the lack of interoperability among components. Software developers have responded to the growing need for interoperability by devising integration solutions. Unfortunately, the task of selecting a suitable solution is difficult. This thesis addresses this problem by making integration a design decision through the creation of an integration taxonomy. This taxonomy is based on *integration elements* that embody the core functionality needed to connect interoperating components.

ACKNOWLEDGEMENTS

I would first like to thank Dr. Gamble for being my thesis advisor and most importantly for guiding and encouraging me throughout my graduate career. I would like to thank Dr. Sheno and Dr. Redner for taking time out of their busy schedule to read my thesis and for being on my thesis committee. I would also like to thank everyone in the Software Engineering Architecture Research office, especially Jamie Payton, for their support and encouragement throughout the thesis process. Last and not least I would like to thank Mark Berryman for believing in me.

TABLE OF CONTENTS

ABSTRACT	III
ACKNOWLEDGEMENTS	IV
TABLE OF CONTENTS	V
LIST OF FIGURES	VI
CHAPTER I	1
1. INTRODUCTION	2
CHAPTER II	7
2. BACKGROUND	8
2.2 Off-the-Shelf Middleware	9
2.3 Design Patterns	10
2.4 Software Architecture	10
2.5 Automation Attempts	11
2.6 Other Frameworks	13
2.7 Development Life Cycles	14
CHAPTER III	18
3. ARCHITECTURE INTEGRATION ELEMENTS	19
3.1 Translator	19
3.1.1 <i>The Minimum Requirements for a Translator</i>	20
3.1.2 <i>Data Handling during Translation</i>	21
3.1.3 <i>Handling Multiple Input Ports for Translation</i>	22
3.1.4 <i>Handling Multiple Output Ports</i>	23
3.1.5 <i>Internal or External Storage for use by Conversion Relation</i>	24
3.1.6 <i>Bi-directional Translation</i>	24
3.2 Controller	25
3.2.1 <i>The Minimum Requirements for a Controller</i>	26
3.2.2 <i>Decisions based on Input Data</i>	27
3.2.3 <i>Decisions based on Input Components</i>	27
3.2.4 <i>Decisions based on output components</i>	28
3.3 Extender	28
3.4 Potential Integration Architectures	29
3.4.1 <i>Translators in a Series</i>	29
3.4.2 <i>Coordinating Input to Multiple Translators</i>	30
3.4.3 <i>Coordinating Output to Multiple Translators</i>	31
3.4.4 <i>Extending the Functionalities in a Integration Architecture</i>	32
3.4.5 <i>A Complex Integration Architecture</i>	33

CHAPTER IV	35
4. THE TAXONOMY OF CORRESPONDENCE IDENTIFICATION	36
4.1 Integration Strategies that contain Translators	40
4.1.1 <i>Bridge</i> [GHJV95]	40
4.1.2 <i>Adapter</i> [GHJV95].....	41
4.1.3 <i>Filters</i> [Lea94]	42
4.1.4 <i>Converters</i> [Lea94].....	42
4.2 Integration Strategies that contain Controllers	43
4.2.1 <i>Façade</i> [GHJV95]	43
4.2.2 <i>Blackboard Controllers</i> [GSP99, Butler98]	44
4.2.3 <i>Rulebased Controller</i> [SGa97, Butler98]	44
4.2.4 <i>Sequentially Composed Filter</i> [Lea94].....	45
4.2.5 <i>Dynamically Composed Filter</i> [Lea94]	45
4.2.6 <i>Conditionally Composed Filter</i> [Lea94]	45
4.2.7 <i>Selector Filter</i> [Lea94]	46
4.2.8 <i>Weighted Filter</i> [Lea94].....	46
4.3 <i>Extender</i>	46
4.4 Integration Strategies that contain Translators and Controllers	46
4.4.1 <i>Mediator</i> [GHJV95]	47
4.4.2 <i>Database Gateway</i> [BS95].....	47
4.4.3 <i>Application Gateway</i> [BS95]	48
4.4.4 <i>Action Control Filter</i> [Lea94]	48
4.4.5 <i>Server-Legacy Wrapper</i> [Cole97].....	49
4.5 Integration Strategies that contain Translators and Extenders	49
4.5.1 <i>Proxy</i> [BMRSS96, GHJV95].....	50
4.5.2 <i>Wrapper</i> [Mularz94].....	51
4.5.3 <i>Decorator</i> [BMRSS96, GHJV95].....	51
4.5.4 <i>Sequential Filter</i> [Lea94]	52
4.5.5 <i>Reducer</i> [Lea94]	52
4.5.6 <i>Differencer</i> [Lea94].....	52
4.6 Integration Strategies that contain Controllers and Extenders	53
4.7 Integration Strategies that contain Translators, Controllers and Extenders	53
4.7.1 <i>Broker</i> [BMRSS96, Mularz94]	53
4.7.2 <i>Client-Server Broker</i> [Cole97]	54
4.7.3 <i>Guidance Comparator</i> [Lea94]	54
4.7.4 <i>Work Flow Manager</i> [Mularz94]	55
4.8 <i>Loosely Defined Integrated Strategies</i>	55
CHAPTER V	57
5. INTEGRATION ARCHITECTURES OF MIDDLEWARE	58
5.1 CORBA	58
5.2 DCOM	61
5.3 Java RMI.....	62
5.4 JavaBeans.....	63

5.5 MQIntegrator-----	64
5.6 Component Broker -----	66
5.7 Polyolith-----	67
CHAPTER VI -----	69
6. CONCLUSION AND FUTURE WORK-----	70
CHAPTER VII -----	72
7. REFERENCES -----	73

LIST OF FIGURES

Figure 1: Integrating Component Architectures Process.....	16
Figure 2: A Minimal Translator.....	21
Figure 3: Chunk to Stream.....	22
Figure 5: A Translator with Multiple Input Ports.....	23
Figure 6: Translator with Multiple Outports.....	24
Figure 7: A Translator with Internal or External Storage.....	24
Figure 8: The Minimum Requirements for a Controller Integration Element.....	27
Figure 9: Controller based on Input Component.....	28
Figure 10: Controller based on Output Components.....	28
Figure 11: Translators in a Series.....	30
Figure 12: Controller to many Translators.....	30
Figure 13: Many Translators to a Controller.....	32
Figure 14: Extending the functionality of the Translator.....	33
Figure 15: A Complex Integration Architecture.....	34
Figure 16: A Taxonomy of Potential Solutions for Architectural Integration.....	39
Figure 17: CORBA in the Integration Taxonomy.....	60
Figure 18: CORBA's Integration Architecture.....	60
Figure 19: DCOM in the Integration Taxonomy.....	62
Figure 20: Java RMI in the Integration Taxonomy.....	63
Figure 21: JavaBeans in the Integration Taxonomy.....	64
Figure 22: MQIntegrator in the Integration Taxonomy.....	66
Figure 23: Component Broker in the Integration Taxonomy.....	67
Figure 24: Polyolith in the Integration Taxonomy.....	68

CHAPTER I
INTRODUCTION

1. Introduction

The demand for cost efficient and reliable software, along with increasing competition, is forcing companies to re-focus their software development practices. Many developers are familiar with constructing a system from scratch, which can be extremely expensive and take a prohibitive amount of time to complete. In order to reduce costs and time-to-market while producing reliable and well-tested software, developers are now turning towards Component-Based Software Engineering (CBSE). CBSE is the study of building a new system or migrating an existing system using heterogeneous components that include commercial off the shelf products (COTS), government off the shelf products (GOTS), and in-house development

Using heterogeneous components, integrating systems with COTS products, and migrating systems to new application environments are new concepts to many software developers. Implementation of an integrated system can uncover problems that come as an unwelcome surprise because the participating components do not interoperate as expected. Comer defines interoperability as the “ability for diverse computing systems to cooperate in solving computational problems” [Comer95].

Interoperability problems encountered during implementation can be attributed to many factors. Developers do not yet have a rigorous methodology to follow when building a system using existing components and therefore may make unwise choices concerning which components to use and how they interoperate. Such a methodology must require integration to be a design decision. If the implications of integration are considered early, when formulating the overall system, decisions can be made as to

which products best interoperate and what design decisions need revising to lessen the complexity of the integration. By systematically justifying these decisions, solutions to interoperability can be traced back to the initial design, allowing for reusability and evolvability as participating components change.

Software developers have responded to the growing need to solve common problems by producing a variety of solutions such as design patterns and middleware, and software architecture. Design patterns (e.g. Adapter, Bridge, Broker, Proxy) identify common implementation problems and provide a design or implementation solution to that particular problem [GHJV95, BMRSS96]. Some of these patterns are applicable to resolve integration problems. A drawback is that it can be a formidable task to discover the correct solution to interoperability problems since there often seem to be several similar solutions to a particular problem. Often the pattern's problem statement is not stated in the terms that makes it immediately identifiable as an appropriate solution. Therefore, a significant amount of time and effort may be spent to resolve problems that arise during integration.

Off-the-shelf (OTS) middleware (e.g. Java RMI, JavaBeans, CORBA) are integration solutions that provide developers with software to help solve interoperability conflicts during implementation. However, in many instances, these solutions lack standardization, and are overly complex or incompatible which makes them difficult to use [Charles99, DMT99]. Therefore, these solutions can be difficult or even impossible to modify during system upgrade or system evolution since the underlying cause of the problem cannot be traced.

Perry and Wolf describe software architecture as a “set of architectural elements that have a particular form” [PW92]. Software architecture describes the foundation of a system and provides the knowledge of how the system works [SG95]. One possibility to predicting significant interoperability problems is to evaluate all of the systems components at the software architectural level before combining them into an integrated system [AbdAllah96, BCTW96, GAO95, Sitarman97] because problems found at the system’s architectural level will cause problems at the implementation level. This will force integration to be a design decision rather than an implementation or technology decision.

Given that architectural analysis can detect potential interoperability problems [GAO95, AbdAllah96, Gacek97, Sitarman97] design decisions can be made early to determine the underlying architecture of the middleware, the complexity of the integration, the effect on new design to ease interoperability problems and facilitates the choice of COTS products. Thus, it can also reduce the cost and increase the stability of the system being integrated by eliminating ad hoc methods of integration solutions. Unfortunately, software architecture lacks key resources to aid developers during the integration process such as:

1. a set of normalized architecture characteristics that deter interoperability
2. a process that analyses the architectures characteristics and predicts conflicts among them
3. connector models whose core functionalities are defined at the same abstract level as software architecture that can resolve interoperability problems

assessed through the architectural characteristics

4. connectivity between the connector models and published integration strategies
5. a link between the connector models, integration strategies and commercial middleware, therefore moving the solution from the architectural level to the implementation level
6. a process that guides the developers through the integration process

This thesis focuses on the connector models, their connectivity to integration strategies and, a link between the connector models, integration strategies and commercial middleware.

By abstracting the core functionalities of integration strategies and the published interoperability problems, three *integration elements*: translator, controller and extender have been identified. These elements are at the same abstraction level as software architecture but form a connection to solutions at the implementation level. The translator and controller elements are indivisible connector models of integration functionality that are based on solving incompatibility problems among components. The extender is a connector model whose functionality enhances the functionality provided by translators and/or controllers. Interoperability problems often require more than one integration element as their solution. Composing sets of these integration elements forms an *integration architecture* which forms the foundation of a taxonomy consisting of common integration strategies.

By developing a taxonomy, we display a common denominator between implementable integration solutions and integration elements. The taxonomy provides developers with a tool for clarifying the intention and capabilities of integration strategies and off-the-shelf middleware. It also forms a crucial connection between actual implementable integration solutions and the abstract architectural level of integration strategies.

This thesis is organized into the following chapters. Chapter 2 presents research that is directly relevant to this thesis. In Chapter 3, the integration elements are defined. Chapter 4 describes the integration elements and relates them to integration strategies using the taxonomy. In Chapter 5 we discuss links between off-the-shelf middleware and the taxonomy. Finally, Chapter 6 concludes with final results and future work.

CHAPTER II
BACKGROUND

2. Background

Researchers have responded to problems in CBSE by producing a variety of solutions to aid in solving common interoperability problems. These methods vary from solutions that can be used at the implementation level to solutions at the architectural level. However, the research areas and goals from which these solutions arise are very different. Lack of uniformity contributes to the difficulty of defining an all-encompassing framework or process to address integration problems. We will briefly discuss the following relevant research development areas.

- Off-the-shelf Middleware – Off-the-shelf middleware is commercial integration software that can be used to solve interoperability conflicts.
- Design Patterns - Design patterns are experience-based descriptions that identify common problems and provide a design or implementation solution to that particular problem [BMRSS95, GHJV95].
- Software Architecture – Software architecture is the study of the elements that identifies their generic behaviors, characteristics and connectivity in a system at an abstract level [SG96].
- Automation attempts - Attempts have been made to produce an automated process that can predict interoperability problems at the architectural level from the definition of a system's architecture [AbdAllah96, Gacek97, Sitarman97].

- Integration Frameworks –Researchers are trying to develop a framework that underlies an architecture [Deline99, Butler97].
- Development Life Cycles – Recent life cycles are available for CBSE which define the development process of a software system from its inception to its deployment.

2.2 Off-the-Shelf Middleware

OTS middleware products are commercially available software products used to integrate complete applications throughout a network or a distributed program, allowing clients access to products transparently [Charles99]. Recently, the popularity of middleware has grown as an increasing number of companies merge or acquire smaller companies, or upgrade their infrastructure. In spite of its growing popularity, middleware is still difficult to use and manage because the solutions are not standardized [DMT99, Charles99]. OTS middleware is chosen during implementation, which often results in implementation difficulties. Problems occur because developers are not familiar with recognizing interoperability problems, and therefore are uncertain how to utilize the middleware to resolve the interoperability problem. Because middleware is used for component integration it must be flexible and evolvable when:

- Components are upgraded or modified.
- The component-based system must integrate with another integrated system.

Difficulties in the above areas for middleware usage are still apparent [DMT99, Charles99].

In order for OTS middleware to become the interoperability solution of the future, more research must be performed to increase its flexibility and adaptability, and to facilitate implementation. In Chapter 5 we analyze several middleware solutions and describe their underlying integration functionality.

2.3 Design Patterns

"Patterns help you build on the collective experience of skilled software engineers. They capture existing, well-proven experience in software development and help to promote good design practice" [BMRSS96]. Design patterns increase reuse by improving a system's flexibility, extendibility, and portability [GHJV95]. There are patterns that contribute integration solutions for CBSE, but few are categorized by those objectives. Another disadvantage of design patterns is that they provide solutions to a specific problem. Therefore, a developer may find many similar patterns that can solve their problem, but finding the ideal pattern is difficult especially if the developer does not understand the core integration functionalities of the design pattern. In this thesis, we examine several design patterns and deduce which patterns can be called integration patterns. We describe these patterns in terms of their underlying integration functionality.

2.4 Software Architecture

Software architecture can be defined as a "description of elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on these patterns" [SG96]. A system's software architecture should be described during the design of a system because the importance of understanding a system's architecture grows as the size and complexity of a system

increases [SG96]. No amount of excellent programming can overcome a bad architecture design. This is because information found at the software architecture level affects the outcome at the implementation level [BMRSS96]. Therefore, understanding a system's architecture is often the first step needed when designing a completely new system, and should also be the first step used when designing a system using existing components, COTS, and GOTS products. An advantage of examining preexisting components that are needed to build an integrated system at the architectural level is that certain clashes or "mismatches" among specific architecture characteristics can be detected [KG99].

2.5 Automation Attempts

The need for a solution that can aid in solving interoperability problems has led to attempts in automating the detection of the architecture mismatches. Thus, these processes rely on research done in software architecture with relation to architectural characteristics, architectural styles, and architectural patterns. Architect's Automated Assistant (AAA) system [AbdAllah96, Gacek97] and Integrating Homogeneous/Heterogeneous Architectures (IHArch) [Sitarman97] are two examples of automated integration analysis tools.

Abd-Allah [AbdAllah96] and Gacek [Gacek97] extended the basic character set [SG96] to run-time issues that distinguish architecture. The AAA system was grounded in descriptions of typical architectural styles, e.g. main-subroutine, and pipe-and-filter. Style characteristics are modeled in Z [Spivey88] in the form of either base elements or conceptual elements. Their base elements are derived from Shaw and Garlan [SG96]. These elements are the common entities that are found in all architectures, such as

components and connectors. The conceptual elements focus on lower level properties such as dynamism, triggering capabilities, concurrency, and distribution. Two disadvantages of AAA are that complete descriptions of the systems being integrated are needed before the use of AAA and AAA does not provide any solutions to the architectural mismatches detected. Abd-Allah [AbdAllah96] and Gacek [Gacek97] compared their mismatches to the ones detected by Garlan et al. [GAO95] during the construction of Aesop. Aesop is a style-generating tool that is constructed from existing systems [GAO95]. AAA did not identify all of the interoperability conflicts, such as conflicts related to event handling and thread of control.

Sitaraman [Sitaraman97] developed IHArch 1.0 as an analysis tool that compared architectural characteristics of independent systems to predict conflicts and to offer potential solutions based on case studies. Sitaraman's [Sitaraman97] characteristics describe the architectures at an architectural level, in which many characteristics cross style boundaries. This level of description resulted in Z [Spivey88] and Object-Z [DKRS91] architecture models [HGKS99, SGa97] that were substantially different than that found in [AbdAllah96, Gacek97]. When exercised, the analysis performed by Sitaraman [Sitaraman97] detected many of the mismatches reported by Garlan et al. during the construction of Aesop [GAO95]. Sitaraman also categorized solutions to those mismatches using a few design patterns. However, these solutions could not be formalized into a generalized integration architecture that would provide an implementation guideline.

2.6 Other Frameworks

Due to the lack of research in building an implementation guideline, several framework building endeavors have been made by Deline [Deline99] and Butler [Butler99].

Independent research by Deline is being conducted to categorize techniques needed to resolve package mismatches during integration [Deline99]. Packaging mismatch refers to conflicts occurring due to differences in the components' interaction mechanisms. Deline states that when reusing components, developers should be concerned about a system's functionality and interaction mechanism and that decisions based on resolving interaction mismatches must be during implementation. Therefore, depending on the conflicts caused by packaging mismatches, a strategy must be found that can resolve the conflicts. Deline's catalog can only be used when conflicts are found during implementation. Depending on the conflict, a suitable strategy must be found that has the ability to solve the packaging mismatch. Waiting to solve interoperability conflicts during implementation has been proven to be very difficult. Therefore, a vast amount of time and code is needed to solve the interoperability problem.

Using the categorization set forth by Sitarman [Sitarman97], Butler's [Butler99] research in integration strategies and patterns led to four categories of integration functions: Translator, Controller, Shared Repository, and Work Flow Manager. The categories were based initially on Mularz's integration architectures [Mularz94], Brodie et al. gateways patterns [BS95] and the adapter, bridge, and façade patterns [GHJV95]. Each category contained abstract features. The foundations of Butler's categories are

based on design patterns, therefore a solution can not be traced to an underlying integration architecture. Using this knowledge, we have expanded and improved upon Butler's initial integration framework.

2.7 Development Life Cycles

Understanding how a system works architecturally is not enough if one wants to build a system that is a success in all aspects [BPEA99]. There are two prominent development processes that can aid the developer in CBSE: the Unified Process [Rational99] and the Model-Based Architecting and Software Engineering (MBASE) lifecycle [BPEA99].

The Unified Process serves as a framework that can be specialized to suit individual needs during CBSE. The process is architecture-centric, and builds the system iteratively and incrementally [Rational99]. It is defined around the four phases of problem solving: Inception, Elaboration, Construction, and Transition [Rational99]. The Unified Process lacks concrete details but provides guidelines for determining interoperability problems and their solutions at all phases of development. In addition, it does not provide resources to build a system using existing components, systems, and COTS products.

The MBASE life cycle is very similar to the Unified Process. Its goal is to bridge issues such as architectural requirements, distributed tasks, achievement of milestones, product costs, product performance and product dependability into a single life cycle. The MBASE lifecycle is divided into four categories of models: Success Models, Process Models, Product Models, and Property Models. The models are worked on consecutively

until the project is complete. There are four milestones in the life cycle, which aid developers in accomplishing their goals. Like the Unified Process, the MBASE life cycle acknowledges the stage at which COTS products and existing software must be chosen and integration plan defined. However, it does not provide a methodology to choose the software that will cause the least interoperability problems, nor does it provide a method to find a suitable integration solution.

The Software Engineering Architecture Research (SEAR) group at the University of Tulsa is researching solutions to interoperability problems through architectural analysis using various complementary approaches, such as formal modeling [HGKS99, SGa97, Butler99], automated architecture analysis [Sitarman97], inheritance of properties, and process definition analysis [PKG99]. Currently, the SEAR group is constructing an interoperability analysis process named Integrating Component Architectures Process (ICAP) [PKG99]. ICAP is a development process that provides the resources needed to evaluate interoperability problems at the architectural level of the participating software components and to derive an appropriate integration architecture for implementation (Figure 1). The goal of ICAP is to supplement the Unified Process and the MBASE life cycle, to form and implement the integration plan

In the pre-integration phase, components, systems, and COTS products that might be used in the integration of the system are identified and characterized. To determine interoperability conflicts, the system's characteristics are compared. Once the conflicts are known, a plan to solve the problems can be discovered and the final choice of a COTS product can be made based on the complexity of the conflicts and their potential solutions. Changes can readily be made to the architectures of systems that are still in the

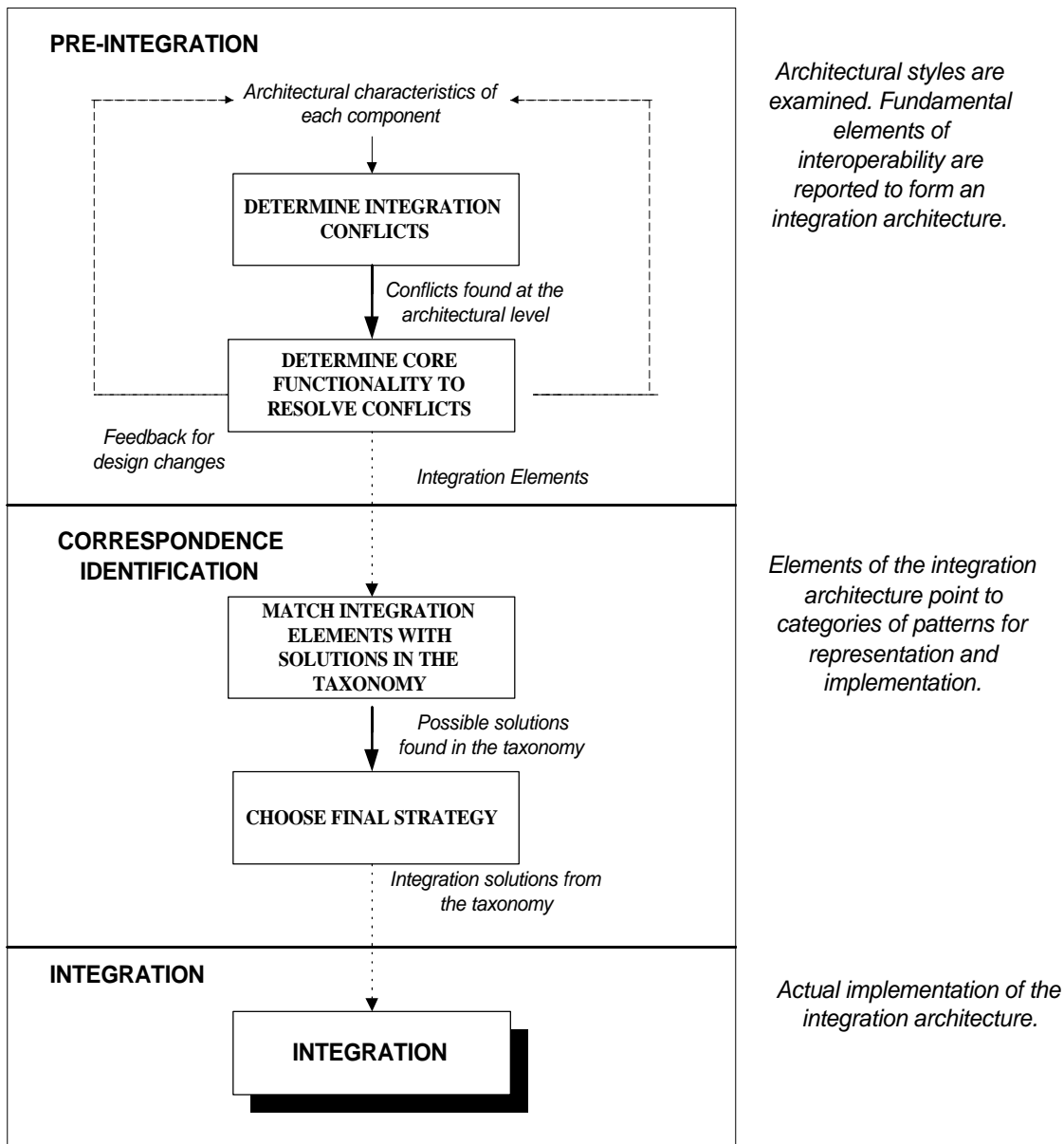


Figure 1: Integrating Component Architectures Process

design phase to avoid predicted conflicts. If changes are made to a system, the developer must iterate through the pre-integration phase again to ensure that the changes did not cause other conflicts.

To complete the pre-integration step, an integration solution at the architectural level is identified. This solution represents an integration architecture composed of integration elements (Chapter 3) defined at the same abstraction level.

Using the integration architecture results from the pre-integration stage a taxonomy of integration strategies is used to choose a suitable design solution (Chapter 4) in the correspondence identification phase. The choice of design solution may depend on implementation platform, available resources, and requirements such as performance and reliability. Upon a final strategy choice, the correspondence identification phase ends. The developer can now begin to incrementally implement the architecture.

Because the underlying architecture is defined, OTS middleware that embodies this architecture can be used for implementation (Chapter 5). This choice is traceable back to design decisions. The ICAP integration phase begins by implementing the integration architecture incrementally until an executable architectural baseline is completed. As long as the developer adheres to the possible solutions given through the correspondence identification phase that embody the key integration elements, the architectural foundation needed to resolve existing conflicts will be maintained. However, system architecture design changes will require iteration through the ICAP.

The ICAP methodology helps developers through the integration process therefore increasing the reuse of both integration strategies and existing systems. The integration elements and the taxonomy are only a subsection of ICAP but play an important role in the process.

CHAPTER III
ARCHITECTURE INTEGRATION ELEMENTS

3. Architecture Integration Elements

To resolve interoperability problems in the design of the integrated system of components, it is necessary to define what integration “looks like” from an architectural perspective. In this respect, one could think of integration at this level in the form of elements that provide the “building blocks” to connect the major contributing components. Our purpose in defining these elements was to determine the core functions of these connectors and to model that functionality architecturally. Subsequently, we have defined three *integration elements* called *translator*, *controller* and *extender* and described their connector models. These have evolved from Butler’s modeling research [Butler99], definitions stemming from architecture style modeling [AbdAllah96, AAG95, GSP99, SGa97], case study analysis [Cole97, GAO95, Sitarman97] and the study of integration strategies [BMRSS95, BS95, GHJV95, Lea94, Mularz94]. These integration elements express the primary functional needs for an integration strategy to resolve interoperability problems. Composing the connector models, we can describe integration architectures. Such integration architectures form the underlying foundation of many design patterns and OTS middleware. In the remainder of this chapter, we define the roles of the translator, controller and extender integration elements.

3.1 *Translator*

The first integration element is the translator. The basic function of a translator is to convert data and functions between component formats and perform simple semantic conversions. The translator integration element is an indivisible connector model. Using this information the translator can be formally modeled as an abstract class using Object-

Z [DKRS91].

The section begins by examining the minimum requirements for a connector to be a translator. The need for these requirements is that many strategies perform other functions outside of integration. However, because they satisfy the minimum requirements of a translator, they can potentially be used for architecture integration.

The section continues by showing allowable translator configurations. These configurations can be seen in implementation of translators within the description of integration strategies. Thus, the developer can be more directly guided toward determining the translator to be part of the integration architecture.

3.1.1 The Minimum Requirements for a Translator

To be considered a translator or to embed a translator, an integration architecture must meet the following minimum requirements (Figure 2).

- 1. Port requirements.** The translator has at least one incoming port and one outgoing port. The translator does not need to know the identities of the components sending data to the input ports and receiving data from the output ports.
- 2. Uniform data.** We use the term “data” to refer to procedure calls, signals, parameters, event calls, shared data, remote procedure call and information. The input domain formed by input data off the ports has a uniform structure and/or content that is predefined by the functionality of the translator.

- 3. Converter relation.** A converter is a mathematical relation that performs the translation of the input domain element to the output range. The relation is total, in that all domain elements are mapped to an element in the range, even if the mapping is to error messages or null values. The relation may be a composition of relations. If there are multiple mappings per input, then the converter relation would provide the entire relational image as output. Decisions on which output to use would require an additional component in the integration architecture (See Controller Section 3.2). If it is a requirement of the translator to produce a single mapping instance per domain element, then the converter relation must be restricted to a total function.

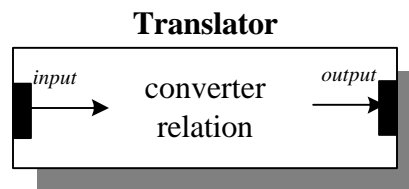


Figure 2: A Minimal Translator

3.1.2 Data Handling during Translation

The converter relation of a translator may have a predefined way in which it handles data. For example, it may process an input stream of data to an output stream of data (Figure 2). It may take a “chunk” of data, such as a sequence, and process it to another “chunk” of data, such as a set. In this case, the domain of the converter relation is a set of sequences of data. It may take a “chunk” of data and pull out individual elements to output a flat stream (Figure 3). In this case, the domain would be formed by the

elements to be individually processed. A translator can also modify parameters and names of function calls across architectural boundaries, as long as the modification is consistent. It may also take in a stream, apply a delimiter to produce a “chunk” of data, in which the domain is formed, by the “chunks” (Figure 3). For example it may change incremental data into discrete data by placing a delimiter on the receipt of the incremental data, which when achieved places the data into a single packet to submit as discrete output. The restrictions placed on this processing are that it cannot be based on the value of the data, it must be predefined and static.

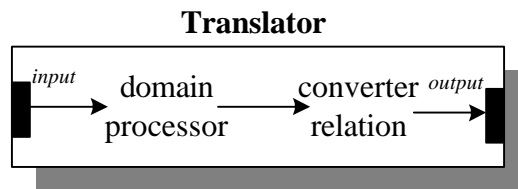


Figure 3: Chunk to Stream

3.1.3 Handling Multiple Input Ports for Translation

A translator is not restricted to a single input port. Data may be incoming from distinct components. It is required that the data is uniform in the domain for translation. If the data is uniform across all ports, then a composition procedure would be employed to determine how to combine the data to appear as if coming from a single source (Figure 4). The composition procedure should be predefined and static and it should not be based on which components are sending the data or even the data itself. These types of decisions are made by a controller integration element (Section 3.2). For instance translator composition procedure may be based on first in first out (FIFO) or load

balancing or fairness across the translator's internal ports. The constraint that all of the input data is placed in the domain must also be satisfied by the composition procedure.

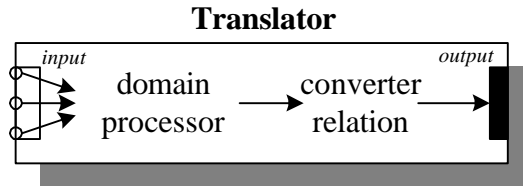


Figure 4: A Translator with Multiple Input Ports

In the case where the data is not uniform across all ports, then the converter relation may use a domain that is a set of tuples. Input data from distinct ports would be placed in predefined domain tuples. For example, if on Port 1 the data is of type Integer and on Port 2 the data is of type Character and on Port 3 is of type Sequence of Integer, it is expected that the converter relation accepts as input from the domain an ordered tuple of type (Integer, Character, Sequence of Integer), such as (1, A, <5,4>). The composition procedure is used to combine the input into the accepted tuple format for translation. For instance, null values may be allowed in any tuple position depending on the composition algorithm.

3.1.4 Handling Multiple Output Ports

The converter relation produces a single stream of output (Figure 5). In the case where there are multiple output ports, this stream of data is reproduced to all output ports. This restriction is due to the lack of decision making power of the translator.

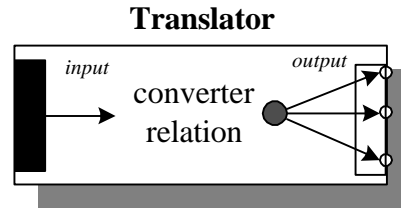


Figure 5: Translator with Multiple Outputs

3.1.5 Internal or External Storage for use by Conversion Relation

There are cases where the converter relation is in the form of a look-up-table, e.g., a name service. Storage for sheer information may be either part of the internal make-up of the translator or externally available and possibly shared with other components (Figure 8). For example the function call from system 1 to system 2 may be **foo(param1, param2)**. If system 2 can only accept the first parameter, the translator would pass **foo(param1)**.

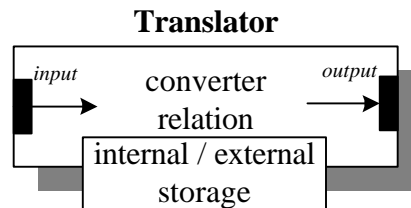


Figure 6: A Translator with Internal or External Storage

3.1.6 Bi-directional Translation

It is possible for a translator to be formally modeled as bidirectional. To achieve bidirectionality, the converter relation must be a total, bijective function. By being total, it fulfills the requirement that all of the input can be mapped to some output. By being a function, it restricts an input to be mapped to at most one output. A bijective function

means that the inverse is also a total function.

There is a problem with depicting a translator as being bidirectional using the model we have described. We model the translator with data being received on its input ports and being passed to its output ports. For it to be bidirectional, the port directionality would be reversed, which is not accommodated by the model. Thus, bidirectionality may be better served by the use of two separate translators.

3.2 Controller

A *controller* integration element coordinates and mediates the movement of information between components using predefined decision-making processes. The decisions include determining

1. which data is passed forward,
2. from which components to accept data,
3. to which components to send data, or
4. a combination of the above decisions.

If decisions are made with respect to components, then the controller needs to know the identity of those components. A data store is allowed for the decision making strategies. The controller must maintain the information in its original incoming format. Format changes require a translator. Variations and specializations of this model can be found in [HGKS99, SG97]. In this section, we define the connector model for the controller integration element. The section continues by describing allowable controller

configurations.

3.2.1 The Minimum Requirements for a Controller

To be considered a controller or to embed a controller, an integration architecture must meet the following minimum requirements (Figure 7).

- 1. Port requirements.** The controller has at least one inport and one output. Depending on the decision-making strategies employed, a port may need to know the identity of the component(s) to which it connects.
- 2. Decision making strategies.** The controller must have at least one strategy or rule on which it bases its decisions. It may have a set of decision-making strategies from which to choose one or a combination of strategies to be applied at once.
- 3. Decision making algorithm.** A controller has a decision-making algorithm that determines the strategy to be employed and applies it accordingly. The strategies may be static, once chosen, or dynamically changing throughout execution. These strategies may be stored internally or externally with respect to the controller component. The useable strategies may change according to the data, components contributing or receiving data and or state of computation. Differences among controllers are based on the manner in which decisions are made. We detail these distinctions in the remainder of the section.

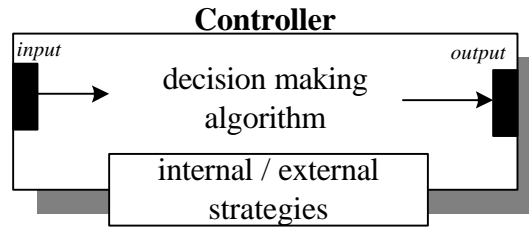


Figure 7: The Minimum Requirements for a Controller Integration Element

3.2.2 Decisions based on Input Data

Input data to a controller can be in any predefined, acceptable format. The content, format, or quantity of the data may influence the controller to choose which information to accept and to pass through to output.

3.2.3 Decisions based on Input Components

A controller may have strategies that determine when an input component can submit data. For instance, these strategies may be fairness or priority related. If a controller makes decisions based on its input components, it must have some awareness of those components as they relate to the input ports of the controller (Figure 8).

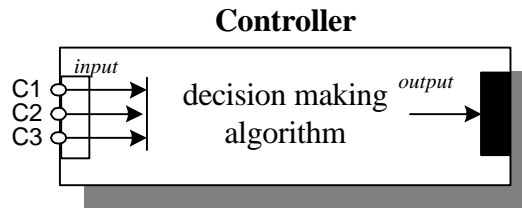


Figure 8: Controller based on Input Component

3.2.4 Decisions based on output components

The controller may have decision-making capabilities to identify to where its information is passed. In this situation, the controller must be aware of those components that are connected to its output ports. For example, decisions may be based on the data content, format and/or quantity. In addition, fairness or priority of components may play a role in decision-making (Figure 9).

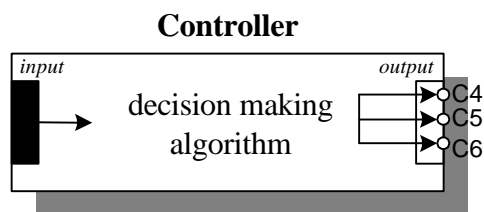


Figure 9: Controller based on Output Components

3.3 Extender

An *extender* integration element adds new features and functionality to a component to adapt it to its behavior within the integrated system. Its functionality can vary extensively. It is specifically used as part of an integration architecture in which the translator and controller cannot accommodate the full functional need.

An extender may also add functionality to the integrated system as a whole to enhance its overall capability. Depending on the application, knowledge of the components with which the extender interacts may or may not be necessary. An extender does not embody those behaviors that are performed by a translator or controller. Below we identify some specific roles that extenders play to integrate systems.

- house a new shared repository
- add a call-back for communication between opportunistic components and direct call-and-return components
- add hand-shaking or buffering of information between two components
- perform mundane tasks such as opening files, closing files and security checks
- creation and deletion of memory blocks

3.4 Potential Integration Architectures

In this section we will highlight some of the potential integration architectures that are needed to solve interoperability problems.

3.4.1 Translators in a Series

Multiple translators in a series can be composed when the data needs to go through several translations (Figure 10). A single integration architecture could be built with the ability to perform the jobs of the individual translators while maintaining a process that is simple and easy to compute [Lea94]. The Converter filter [Lea94] is an example of an integration pattern whose integration architecture contains several

translators in a series. Each translator will have its own converter and will function independently. The functionality of the Converter is discussed further in Chapter 4.

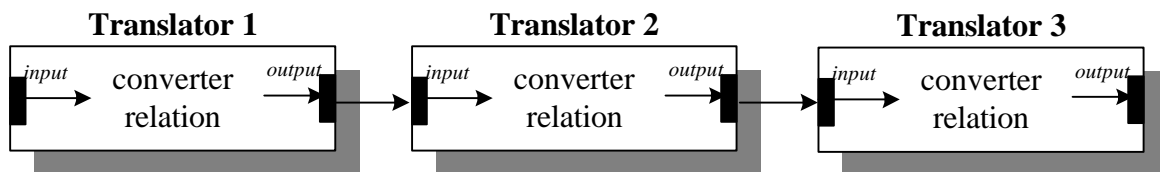


Figure 10: Translators in a Series

3.4.2 Coordinating Input to Multiple Translators

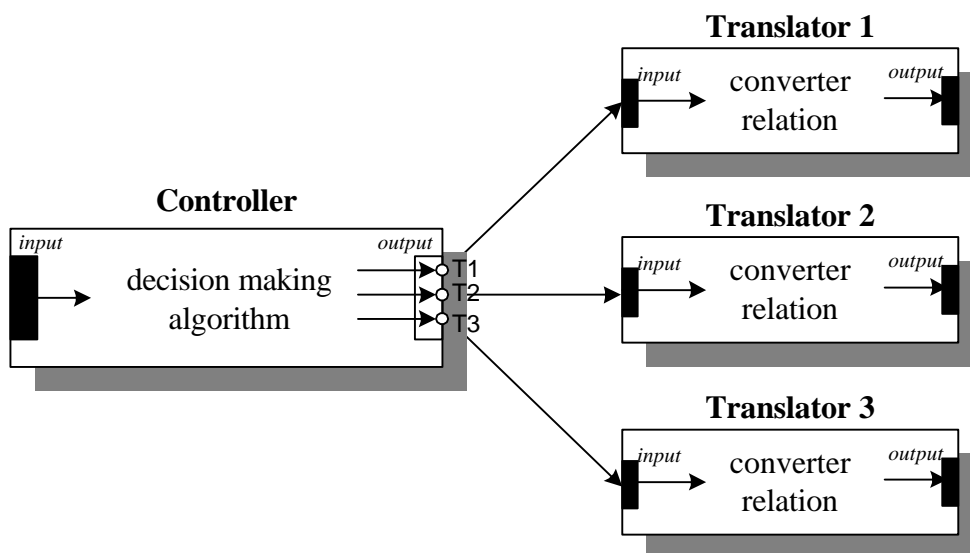


Figure 11: Controller to many Translators

Since translators do not have decision-making capabilities, a controller is needed to coordinate data flow when there are several translators in an application (Figure 11). For instance, the mediator pattern [GHJV95] is responsible for controlling and coordinating the interactions of a group of components. Its integration architecture

consists of a controller and many translators. The controller calls the appropriate translator based on the incoming data. The translator integration element will then perform the appropriate conversion for communication with the outgoing components. The functionality of the mediator pattern is further discussed in Chapter 4.

3.4.3 Coordinating Output to Multiple Translators

Another potential integration architecture is that of a controller following a translator to filter results to the appropriate components. (Figure 12). For example the Bridge pattern is used when several different implementations of a component needs to uses the same class abstraction (see Chapter 4) [GHJV95]. Since the integration architecture Bridge pattern consists of a translator, the developer needs to implement a controller integration element when explicit component invocation is required. On the other hand, the integration architecture of a database gateway pattern contains both a translator and a controller [BS96]. The database gateway is used when there are many applications that need to use a database. After the data has been translated the controller will invoke the corresponding output components based on the needs of the input components.

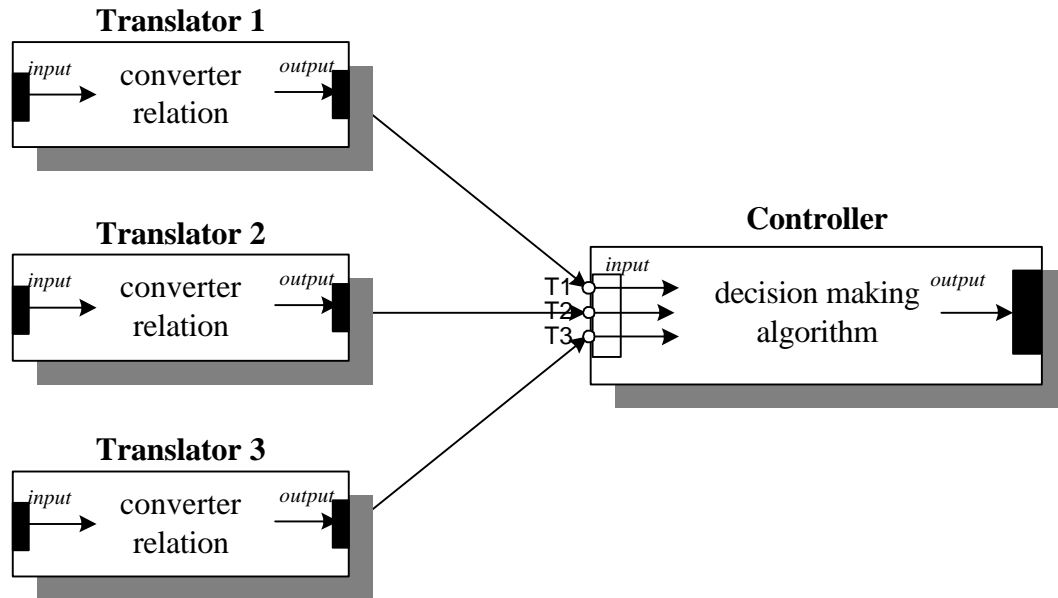


Figure 12: Many Translators to a Controller

3.4.4 Extending the Functionalities in a Integration Architecture

An extender integration element is needed when the data needs to be processed before or after translation. It can also be used to sort data, open or close files, and as a temporary buffer. For example the proxy is often used to defer the full cost of creating and initializing an object until the object is actually needed [GHJV95]. The proxy pattern's integration architecture contains a translator and an extender. Depending on the application the functionality of the extender can vary. In this example (Figure 13) a proxy is needed before the translator to sort the incoming data from components that do not send their data in the form expected by the translator. This allows the translator to convert functionality simply and decreases its processing time. The proxy pattern is discussed further in Chapter 4.

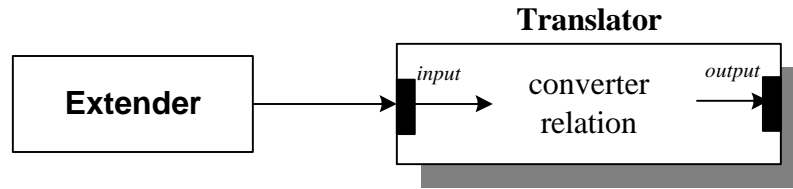


Figure 13: Extending the functionality of the Translator

3.4.5 A Complex Integration Architecture

A combination of a translator, controller, and extender is needed when all of the integration elements functionality is needed to solve the integration problem. For example integration architecture in off-the-shelf middleware CORBA contains translators, controller and extenders (Figure 14). The controller is needed to make decisions concerning issues like which server is available and which server contains the functionality needed to perform the required task. The extender in CORBA often performs security checks and will only allow authorized clients to pass requests to the controller. The translators are needed to translate incoming data into a form that the controller expects. These translators follow the logic of the Adapter translator (see Chapter 4). Basically a combination of all three elements can solve all the interoperability problems faced by the developer.

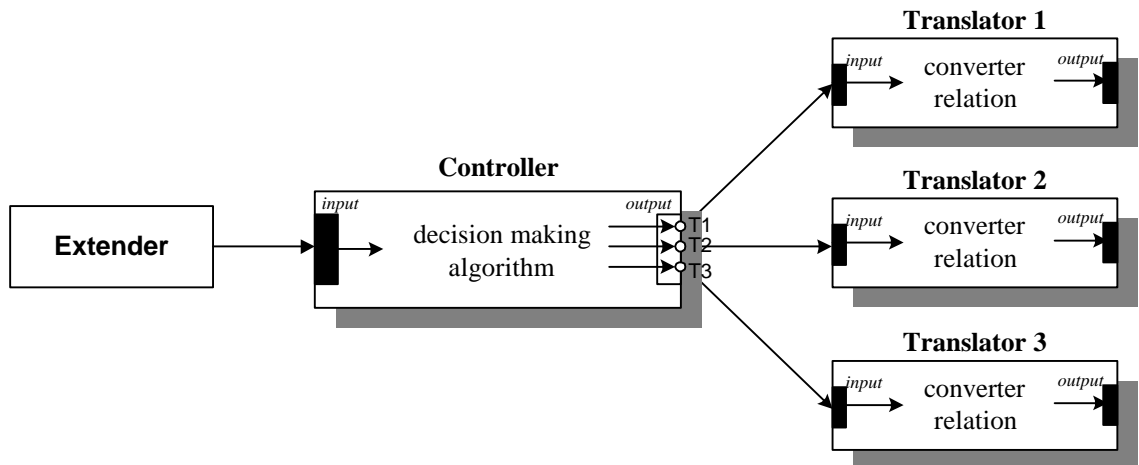


Figure 14: A Complex Integration Architecture

CHAPTER IV
THE TAXONOMY OF CORRESPONDENCE IDENTIFICATION

4. The Taxonomy of Correspondence Identification

As we have shown in the previous chapter, the integration elements are modeled to provide a baseline of observable, structural behaviors of an integration architecture to resolve interoperability problems. An integration taxonomy is needed to help developers transition from the integration solutions they find at the integration architectural level to solutions at the implementation level.

Using literature on software component integration, e.g., [AbdAllah96, Gacek97, Sitarman97], and patterns, e.g., [BMRSS96, BS95, GHJV95, Lea94, Mularz94], we form a partial integration taxonomy (Figure 15) that shows the relationship between the architectural integration elements and the solutions at the implementation level [KG98]. We highlight each integration strategy and give details as to why its functionality places it in a particular position in the integration taxonomy. We use the term *integration strategy* to encompass integration processes, techniques, solutions, gateways, and patterns. We consider an integration strategy for the taxonomy if it satisfies one or more of the following criteria:

- it is a published solution and embodies functionality that is statically well-defined, or
- states that it resolves an interoperability problem, or
- states that it is related to an accepted integration strategy, or
- describes one or more mechanisms to decouple or increase functionality, or

- develops a new interface for a component.

By analyzing the integration strategies we are able to categorize them according to their main functionalities, resulting in the current structure of the taxonomy. While the integration strategies placed in the taxonomy may resolve particular interoperability problems, they are not restricted to having additional functionality and implementation considerations outside of integration. Their placement in the taxonomy means that they embed one or more models of integration functionality.

The integration taxonomy enables developers to make integration decisions at the design stage and provides them with logical connections to implementable solutions. It also shows the integration architecture of the final strategies and therefore can be used as a resource to build customized integration solutions.

The integration taxonomy is also useful for developers who do not want to analyze the components to be integrated at the architecture level. Decisions can be made at the implementation level because the taxonomy clearly shows which design patterns can be used for integration and provides concise information about what type of interoperability problems the design patterns resolve. This allows developers to compare and contrast integration pattern as they now have a clearer understanding of the integration patterns functionality.

In the remainder of the section we highlight each integration strategy and discuss how their taxonomy position relates to their integration architecture.

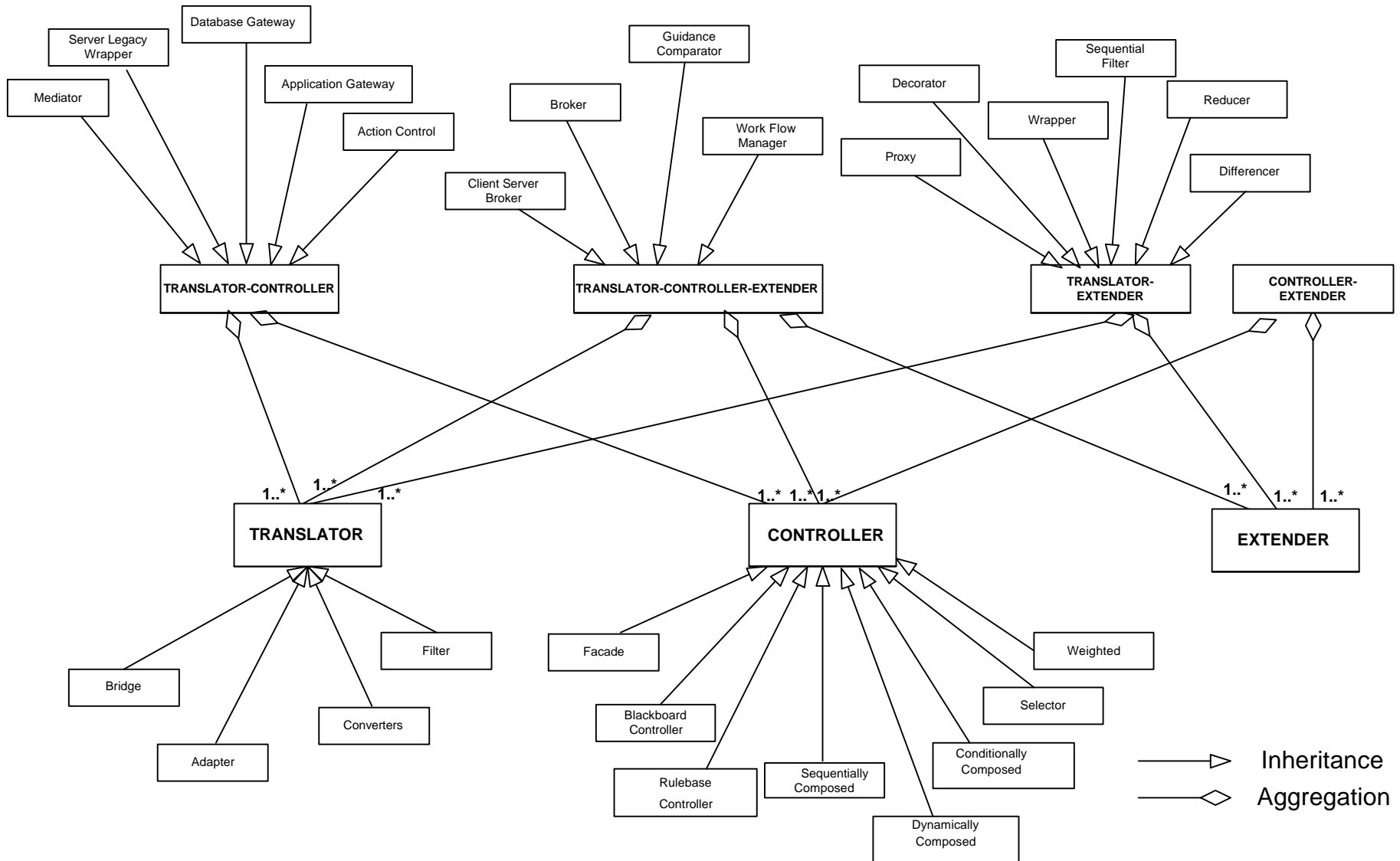


Figure 15: A Taxonomy of Potential Solutions for Architectural Integration

4.1 Integration Strategies that contain Translators

The integration strategies in this section embed one or more translators for integration. They do not include a controller or any functionality that extends the components integration functionality. Typical Translators are Bridge [GHJV95], Adapter [GHJV95], Filter [Lea94] and the Converters described in [Lea94]. In the rest of this section we will show how these patterns fulfill the minimum requirements of the translators.

4.1.1 Bridge [GHJV95]

A Bridge is used when there are several possible implementations that need to use the same abstraction. The bridge decouples the abstraction from its implementation therefore allowing them to vary independently.

An example presented by Gamma et al. [GHJV95] discusses the problems of building a Window abstraction in a user interface toolkit. The toolkit must allow the user to build applications that can execute on a wide variety of platforms. If inheritance is used to build the toolkit it will require the developer to define “Window” each time the application needs to use another platform. A Bridge can solve this problem by separating the Window abstraction from its implementation methods using class hierarchies. Window becomes the root node in the abstraction hierarchy and WindowImp becomes the root node in the implementation hierarchy. Hence the “one to one relation” between WindowImp and Window is called the Bridge [GHJV95].

The bridge in this pattern receives generic commands sent by the abstract class,

which satisfies the translator rule that requires the translator to receive uniform data. All incoming data is translated since the abstract class “forwards clients requests to the implementation objects” via the Bridge [GHJV95]. Because of the 1-1 relationship the Bridge has a least one input port and output port. All the implementation methods will receive the result but only one of them will be invoked. In order to pass data to only one implementation method a controller integration element must be added.

4.1.2 Adapter [GHJV95]

An Adapter is used as a data translator between incompatible interfaces that need to cooperate. Gamma et al. [GHJV95] demonstrate the adapter using a drawing editor. The drawing editor consists of graphical objects that can draw themselves and is defined using the abstract class Shape. The abstract class Shape is inherited by classes such as LineShape and PolygonShape. The drawing editor also needs to provide a simple text editor to allow text to be added inside the shape. Instead of rewriting a text editor a product containing a TextView class (which can provide all the required functionalities) is being considered. The integration process has resulted in an interoperability problem because the TextView does not know how to conform to the interface of Shape. Since rewriting TextView code is not a viable option the adapter pattern can be used to solve the problem. An adapter called textShape can be built to adapt TextViews interface to Shapes. The class textShape inherits from Shape but utilizes TextViews interface. For example when textShape receives requests from Shape it translates the input into a form that TextView understands.

The adapter meets the minimum requirements of a translator by having at least

one incoming and outgoing port, receiving uniform input, and having a total function that converts all the incoming input to a form expected by output components. The adapter can also call methods that enhance the ability of the adapter. In this circumstance all methods on the outgoing port will receive data but only the methods that are affected by the data will be invoked.

4.1.3 Filters [Lea94]

Lea [Lea94] defines a “classic” filter as a “pure function that accepts one or more main inputs and returns an output of the same form as the input but with different values” [Lea94]. From his description, we infer that a filter is a stateless function that transforms data from one form to another. From the above definition we can further deduce that the filter has at least one input and one output port and that it also has a total relation between the incoming data and the outgoing data. The filter falls into the translator category that allows for multiple input ports.

4.1.4 Converters [Lea94]

Lea [Lea94] defines the converter as a specialized filter that has the ability to convert data into multiple formats. He explains that something as simple as defining a geographical position can be represented in many different formats as it depends on the coordinate type of the system. The input may need to pass through a series of distinct converters when unified with other representation. The data passes through each translator resulting in a generic representation of the position. Part of the converter’s requirements is for the data to be standardized therefore meeting the uniform data requirements. The converter must also be connected to a source and destination

component which could be another filter. Lea emphasizes that the filters need to be implemented as a simple, freestanding procedure. It is also important to ensure that the filter's output is a function of its input. The converter has met all the minimum requirements of translator. It has added functionality by allowing all incoming data to be translated in multiple forms.

4.2 Integration Strategies that contain Controllers

This section presents integration strategies that contain one or more controllers. They do not have functionality that pertain to either the translator or the extender integration element. A Façade [GHJV95], Blackboard Controller [SGa97, Butler98], Rulebased Controller [SGa97, Butler98], Sequentially Composed [Lea94], Dynamically Composed [Lea94], Conditionally Composed filter [Lea94], Selector filter [Lea94], and Weighted filter [Lea94] are qualified examples of integration strategies that embed the Controller integration element.

4.2.1 Façade [GHJV95]

The Façade embeds a decision making component that has the ability decouple complex systems by providing a simple interface to the subsystem. This simplifies the client responsibility and makes the system easier to use.

Gamma et al. [GHJV95] illustrate the functionalities of a Façade in a complex compiler subsystem. In the example the client needs to use classes such as the Scanner, Passer and the BytecodeStream. Most clients do not need to know how these individual components function. Creating a generic interface (a Façade) between the client and the

components will solve this problem. The Façade will receive requests from clients and then make a decision as to which applicable component to invoke. The data in the Facade is sent to and received from known components in the subsystem components. This allows the developer to build a customize interface without worrying if it is compatible with all the underlying applications.

4.2.2 Blackboard Controllers [GSP99, Butler98]

There are two Controllers in the Blackboard architecture: the distributor controller and the composer controller. The Blackboard's distributor controller has the ability to decide which is the most appropriate knowledge source to interact with the current events posted on the blackboard. Therefore the distributor controller bases its decision upon the output components. The decision strategies used are based on conflict resolution among potentially executing knowledge sources.

The Blackboard's composer controller has the ability to collect changes from the knowledge source and produce a consolidated sequence of changes, which is used to update the Blackboard. Here the decision is based on the input components. Depending on their priorities the composer controller will gather their information to be used to update the Blackboard.

4.2.3 Rulebased Controller [SGa97, Butler98]

The Rulebased Controller is specifically used in the Rulebase architecture. It uses conflict resolution to determine which rule can update the working memory of a Rulebase system. The controller performs a match on the rules precondition based on the contents

of the working memory and then determines what the next executing rule should be based on the input data.

4.2.4 Sequentially Composed Filter [Lea94]

The Sequentially Composed filter has the ability to coordinate the effects of a result to other components by sequentially connecting the outputs of one or more functions as the inputs of others [Lea94]. The use of the term sequentially connecting inputs to outputs suggests integration. The fact that there are multiple outputs coordinated to map to inputs of another function suggest that there is some type of decision making process that performs the mapping.

4.2.5 Dynamically Composed Filter [Lea94]

This filter contains an invocation plan that is used to update available subfilters. From this we can deduce that the Dynamically Composed Filter contains a controller. The controller's decision-making strategy is use conflict resolution to make its decision. Using the incoming data the controller invokes the appropriate rules.

4.2.6 Conditionally Composed Filter [Lea94]

The Conditionally Composed filter “invokes the best available function depending on accuracy and quality estimates of source data” [Lea94]. We can deduce that this is a controller that bases its decision on incoming data. It uses accuracy and quality within its decision-making strategies.

4.2.7 Selector Filter [Lea94]

The Selector has the ability to “select and output the best of multiple inputs” [Lea94]. This suggests that the Selector has a controller. Using information based on the current state of the system the controller decides which is the output among the incoming data.

4.2.8 Weighted Filter [Lea94]

The controller in the Weighted filter has the ability to weigh and combine multiple input based on the accuracy and quality of the incoming data.

4.3 Extender

As we stated in the previous chapter the extender contains additional functionality that is coupled with that of a translator and/or controller. Thus there are no strategies that just fall into this category.

4.4 Integration Strategies that contain Translators and Controllers

This section presents integration strategies that contains at least one translator and one controllers. They do not have functionality that falls under the extender integration element. The Mediator [GHJV95], the Database Gateway [BS95], the Application Gateway [BS95], Action Control filter [Lea94] and the Server-Legacy Wrapper [Cole97] are examples of integration strategies that contain both the Translator and the Controller functionalities.

4.4.1 Mediator [GHJV95]

Building a system from many objects has component reusability in mind but we find that these components rely heavily on each other therefore making it difficult to reuse [GHJV95]. It is also difficult to modify the system due to the tight coupling. The mediator pattern promotes loose coupling by keeping objects from referring to each other explicitly. It is responsible for controlling and coordinating the interactions of a group of distributed objects. Since the mediator is a centralized control unit, translators are needed to convert data from one form to another.

Gamma et al. [GHJV95] described the tight coupling in a graphical user interface between the dialog box and the window, buttons, menus and entry field. One of the requirements is that the options should vary with the selection. (e.g. if a user selects a car as a mode of travel then only the applicable options should be available). The mediator pattern can be used to solve this problem. It can be implemented to accommodate the different options behavior by containing the dependency rules for each option. Decisions are based on the incoming data, which in this example can be something as simple as a pass of control back to the mediator from the objects. The objects only need to know about the mediator. On the other hand the mediator needs to know about all the objects. A translator is needed between an object and the mediator if the object can not communicate with the mediator. This means that there may be several distinct translators in the system.

4.4.2 Database Gateway [BS95]

A Database Gateway is designed to be placed between the application modules

and the legacy database. The Database Gateway's goal is to "encapsulate the entire database from the perspective of the application modules" [BS95]. This prevents the developers from having to alter the application's code. The Database Gateway consists of a mapping table (translator) and a coordinator (controller). The mapping table captures and converts all calls from the application to the legacy database on the server machine. A translator needs to be built to handle complex mapping calls such as one-to-many or many-to-many. It also needs to handle translating the results from the legacy database back to the application. This could in fact be implemented as another translator. It is likely that there will be several databases that the applications needs to use. The Database Gateway must be built to handle to serve applications that utilize one or more applications. The coordinator is used to manage all the database gateway's functions. The coordinator's decisions are based on the needs of the input components.

4.4.3 Application Gateway [BS95]

The application gateway is similar to the database gateway but it is more complex and difficult to build because it is placed further up in the systems architecture. It is designed to be placed between the interface and legacy system. It encapsulates the applications and the database to suit the interface's perspective. The application gateway captures calls from the interfaces and translates them into a form that the legacy system expects. It also coordinates target and legacy updates when the two support or duplicate applications or data.

4.4.4 Action Control Filter [Lea94]

The Action Control Filter calculates the differences between actual and desired

states of a system and finds the value that will minimize the difference. An example, presented by Lea when an engine fails, a special function is needed to compute the best course of action [Lea94]. We can deduce that there is a controller (makes a decision that will rectify the problem) and a translator (performs the actual calculation) in the Action Control Filter. The translator receives the actual state of the system. It will then calculate the difference between the actual state and the desired state and pass the result to the controller. The decision-making strategy in the controller examines the incoming data (the result calculated by the translator) and will invoke a component that can reduce the error.

4.4.5 Server-Legacy Wrapper [Cole97]

The aim of the Server-Legacy Wrapper is to access the legacy components using the same syntax and semantics as the server implementation language. The legacy code acts like a procedure and executes locally within the server space. Building a web-based interface to the legacy code can be difficult. The Server-Legacy Wrapper makes this process easier by using a controller to synchronize the server calls to the legacy calls. Since the legacy system and the interface are written in a different language, a translator is needed to provide communication between them.

4.5 *Integration Strategies that contain Translators and Extenders*

This section presents integration strategies that contain at least one translator and one extender integration element. The strategies do not have a decision-making component. Combinations of the Translator-Extender functionalities are visible in the Wrapper [Mularz94], the Proxy [BMRSS96, GHJV95], Decorator [BMRSS96,

GHJV95], Server Legacy Wrapper [Cole97], Sequential Filter [Lea94], Reducer [Lea94], and Differencer [Lea94] pattern.

4.5.1 Proxy [BMRSS96, GHJV95]

The Proxy is used to translate information and perform pre and post data processing. The proxy is often used to defer the full cost of creating and initializing an object until the object is actually needed. For example opening a graphical object in a document editor is difficult to do therefore taking a long time to open the [GHJV95]. Leaving the creation of the graphical object until after the document is opened can speed up the process.

The proxy can be used as a stand-in for the real graphical image. The proxy pretends that it is the graphical image and only when the image is viewed will it be instantiated. The proxy needs to keep a reference of the image so that it can be called when needed. In this example the proxy has extended the functionality of the system by making the editor more efficient when called. The proxy also contains a translator in that the compressed image can be uncompressed and displayed on the screen. This is just a simple example of what a proxy can do.

The role of the proxy can fall into four categories: remote proxy, virtual proxy, protection proxy and smart reference proxy. The remote proxy extends the systems functionality by being a local representation of the actual object. It contains a translator so that it can interact with all the local objects in the system. The virtual proxy extends the systems functionality by only creating an expensive object on demand. A translator is needed so that the proxy can uncompress and compress the expensive objects. The

protection proxy extends the systems functionality by extending access to the original object. The proxy needs a translator to convert the data types of clients who are incompatible with the original object. The smart reference proxy contains a pointer to where an actual object is situated. This extends the system functionality because it allows the system to be more efficient. The proxy needs a translator when the reference is to an inconsistent data type.

4.5.2 Wrapper [Mularz94]

The Wrapper enables the developer to add new functionality to a legacy system, which needs to work both independently and with other systems. This is accomplished by encapsulating a legacy application so that it is only exposed through a framework interface. New functions that extend the systems original functionalities are also placed in their own framework interface. Since the components are probably incompatible, a translator is needed to provide a mapping between them. It is important that the Wrapper is designed so that both old and new functionalities in the system remain transparent to the user especially since the Wrapper is often used during the transitional period between a new system and an old system. This allows developers to gradually introduce and test a new system without affecting the users.

4.5.3 Decorator [BMRSS96, GHJV95]

A Decorator has the ability to add functionality and additional responsibility to an object dynamically. This is useful when one needs to add responsibility to an individual objects and not to the entire class. When building a user interface using a toolkit it should allow you to add properties such as borders to it [GHJV95].

Additional functionality can be added using inheritance but this can be rather cumbersome. By enclosing the border component with the decorator object a flexible system can be built. The decorator conforms to the borders interface. The decorator forwards requests to the component and performs any additional tasks that can extend and enhance the component's functionalities. The decorator can be recursively added therefore increasing its responsibility but enhancing the system at the same time. A translator is needed since the decorator and its components are not always identical. The decorator pattern is not limited to graphical user interfaces. It can be used when an object needs to be enhanced quickly.

4.5.4 Sequential Filter [Lea94]

A Sequential Filter examines a series of inputs over a certain time period. The data is then averaged. This suggests that an extender integration element is needed to buffer the data over a set period of time. It then passes all the values to the translator. The translator calculates the average and passes the result to the all the outgoing components.

4.5.5 Reducer [Lea94]

By collecting and combining a series of values to a single output the Reducer acts as both an extender and a translator. It extends the functionality of a component by applying weights to inputs, while the translator averages the incoming data.

4.5.6 Differencer [Lea94]

The Differencer transforms incoming data into a single value. A translator is needed to find the difference between the two data values. An extender is needed to

increase the functionality of the translator because it detects when the incoming data's value changes. When this occurs, the extender passes this information to the translator who uses this information to calculate the difference.

4.6 Integration Strategies that contain Controllers and Extenders

We have not yet found any integration strategies that fall into this category. Further research is needed to detect if there are any published integration strategies available that contains the functionalities of only the translator and controller.

4.7 Integration Strategies that contain Translators, Controllers and Extenders

This section presents integration strategies that contain translators, controllers and extender integration elements. The functionalities of a Translator, Controller and Extender are present in the Broker [BMRSS96, Mularz94] pattern, the Client-Server Broker [Cole97], the Guidance Comparator [Lea94], and the Work Flow Manager [Mularz94].

4.7.1 Broker [BMRSS96, Mularz94]

Building a distributed system decouples and increases flexibility but often this causes communication problems. The Broker explicitly coordinates communication between interoperating applications using the controller. The controller in this pattern is actually known as the broker component. The broker component registers server's services and makes them available to clients through an interface. The broker component must know of all the available services in order to pass requests to them. If necessary the

broker component can pass requests to a registered, remote broker. The broker component's decision is based upon the registered services therefore on the output components. The Broker pattern includes a Proxy [BMRSS96, GHJV95]. The Proxy extends the systems functionality by forwarding requests as well as transmitting results and exceptions in either system. The Brokers pattern also contains the Bridge pattern [GHJV95]. The Bridge is needed to translate data between the systems especially in a heterogeneous system (See Bridge Pattern).

4.7.2 Client-Server Broker [Cole97]

A Client Server Broker is composed of a Wrapper [Mularz94] and a Broker [BMRSS96, Mularz94]. The purpose of the Client-Server Broker is to provide a broker between all the components in the system such as a web-based interface and a legacy system. A controller is needed to manage the life cycles of the objects [Cole97]. Adding a component that creates and registers legacy objects further expands the systems functionality. A translator is needed to translate data between the objects.

4.7.3 Guidance Comparator [Lea94]

A Guidance Comparator controls the movement of information between components and ensures that data being passed to the flight director is in an acceptable form and is meaningful. A translator is necessary if the data formats differ between the components. It extends the performance of certain components by categorizing and reducing data into a form that is acceptable to other filters.

4.7.4 Work Flow Manager [Mularz94]

The Work Flow Manager automatically performs user-defined tasks that are based upon repeatable tasks performed by stand-alone components. For example a user uses the same information each time to update the same software products, which is accomplished in the same order. Each product provides indispensable services to the user. A human error is difficult to find and fix and the task can be rather tedious.

The work flow manager can be used to integrate the products that will update the all the application automatically. To achieve this task a managing component is needed (a controller), a data exchange component (a translator) and a component to perform mundane tasks (an extender). The work flow manager contains a controller, translator and an extender. The controller will capture the user input and then invoke the corresponding products based upon the input data. The data will then be translated into the product expects. The extender is needed to locate files and applications, it provides a list of the component profiles and it also can be implemented to add functionality to the over all system. The work flow manager should be used when a problem can be modeled as a set of discrete steps [Mularz94].

4.8 Loosely Defined Integrated Strategies

The Shared Repository pattern [Mularz94], the Information System gateway (IS Gateway) [BS95] and the mediation process [ITU96] are examples of loosely defined integration strategies, such that each could be composed of one or more of the integration elements in Figure 15. The shared repository is an integration pattern that enables separate components to inter-operate while accessing a repository [Mularz94]. The

pattern could be composed of one or more of the following: controller, translator and extender. The IS gateway is used when the developer needs to encapsulate the entire legacy system for integration with outside components [BS95]. The authors' recommend that this gateway is used in all migrations therefore, its definition is flexible. The mediation process can pass information between network groups or between network elements [ITU96]. The process can consist of one or more of the integration elements defined, including a shared repository. While these strategies seem to be part of the taxonomy there are several reasons why they do not fit.

- The inner operability's can vary since the strategies do not explicitly define the required elements. Therefore, depending on the developers' needs they can choose what should be part of the strategy without changing the definition of the strategy.
- The strategy definitions were broad enough that they could include predefined integration patterns as well as the basic elements. For example the IS gateway could included the database gateway or the translator gateway [BS95].
- The loosely defined integrated architectures are all encompassing which makes them hard to fit into the taxonomy. Since the composition of integration elements in the Shared Repository, IS Gateway, and mediation process can vary broadly, the developer can dynamically choose what should be implemented as part of the strategy without changing its initial definition. Thus, these strategies are not represented explicitly within the taxonomy.

CHAPTER V
INTEGRATION ARCHITECTURES OF MIDDLEWARE

5. Integration Architectures of Middleware

With the expansion of companies on a worldwide scale, mergers between corporations, the expansion of a company's network, and the need for distributed systems has increased the demand for simple and fast solutions to interoperability problems. Middleware is that software solution to interoperability problems. Middleware incorporates an infrastructure that supports distributed component-based application development. It provides a means for data exchange and communication among components, hiding distributed information from the developer. Middleware enables a user to integrate heterogeneous systems while ignoring the constraints imposed by the underlying architecture [Purtilo94]. There are many products available such as CORBA, DCOM, Java RMI, JavaBeans, MQIntegrator, Component Broker, and Polyolith that provide these middleware solutions.

This chapter on middleware is an introduction to how these products can be related to the integration strategies and integration elements in our taxonomy. The architectures of middleware solutions consist of several basic components. These basic components are translators, controllers and extenders. Due to the complexity and architectural description ambiguity of middleware, we will only define the obvious integration strategies using the integration taxonomy. Further research on the architecture and integration patterns of middleware is needed.

5.1 CORBA

Common Object Request Broker Architecture (CORBA), developed by the Object

Management Group (OMG), enables software from different vendors to interoperate by allowing programmers to choose the most appropriate application for their system [url2]. CORBA provides integration flexibility and more importantly allows for the integration of existing components [url2]. The CORBA architecture is built on the Broker pattern (see Section 4.7.1) [BMRSS96, Mularz94]. This Broker in CORBA, the object request broker (ORB), is responsible for handling client requests, communication between client and server components, and the separation of a component's interface from its implementation [Vinoski96]. It hides the details of objects, such as their location and implementation, from clients which makes it easier to use. The CORBA architecture also uses the adapter pattern [GHJV95]. The Adapters (Section 4.1.2) in CORBA are responsible for converting CORBA's object adapter into a form that the outgoing components expect. This allows a caller to invoke an object request without actually knowing the object's interface [Vinoski96]. Therefore, CORBA can actually contain multiple adapters. In Figure 16 we see that CORBA architecture consists of the adapter and broker integration strategies which means that all three integration strategies (translator, controller and extender) are present within its integration architecture. We could have mapped the functionalities of the OTS middleware directly to the integration elements. However, since we knew which integration patterns make up the middleware we included them in the diagram to illustrate the connection between the integration elements, integration patterns and middleware.

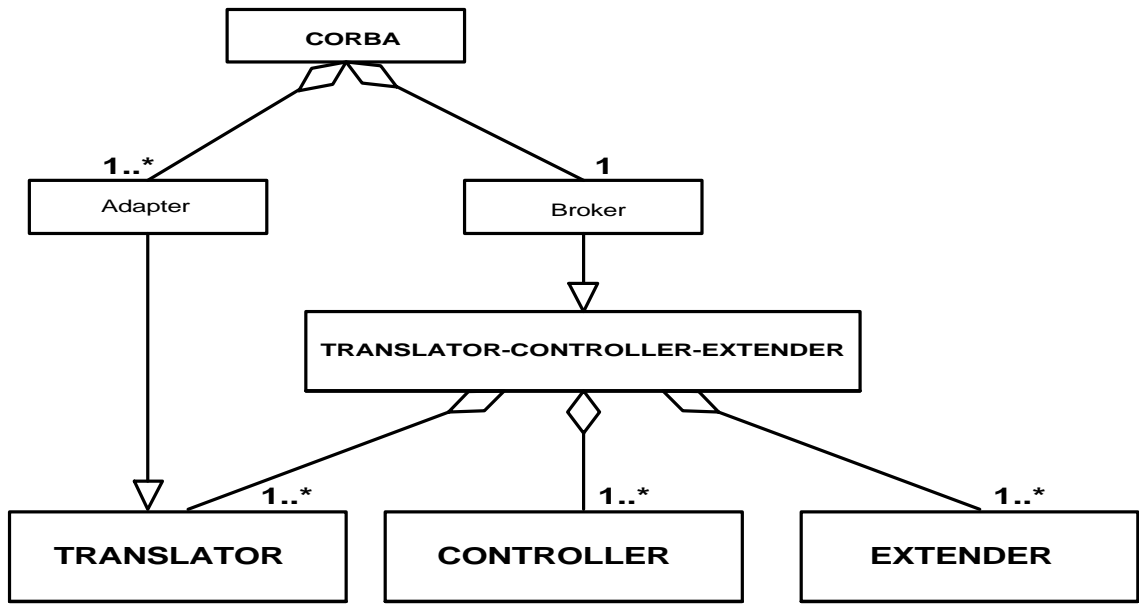


Figure 16: CORBA in the Integration Taxonomy

Figure 17 depicts the integration elements that form the architectures of the broker pattern and adapter pattern. Since the broker and adapter are part of CORBA's architecture we can see what CORBA looks like at the integration architecture level using the integration elements.

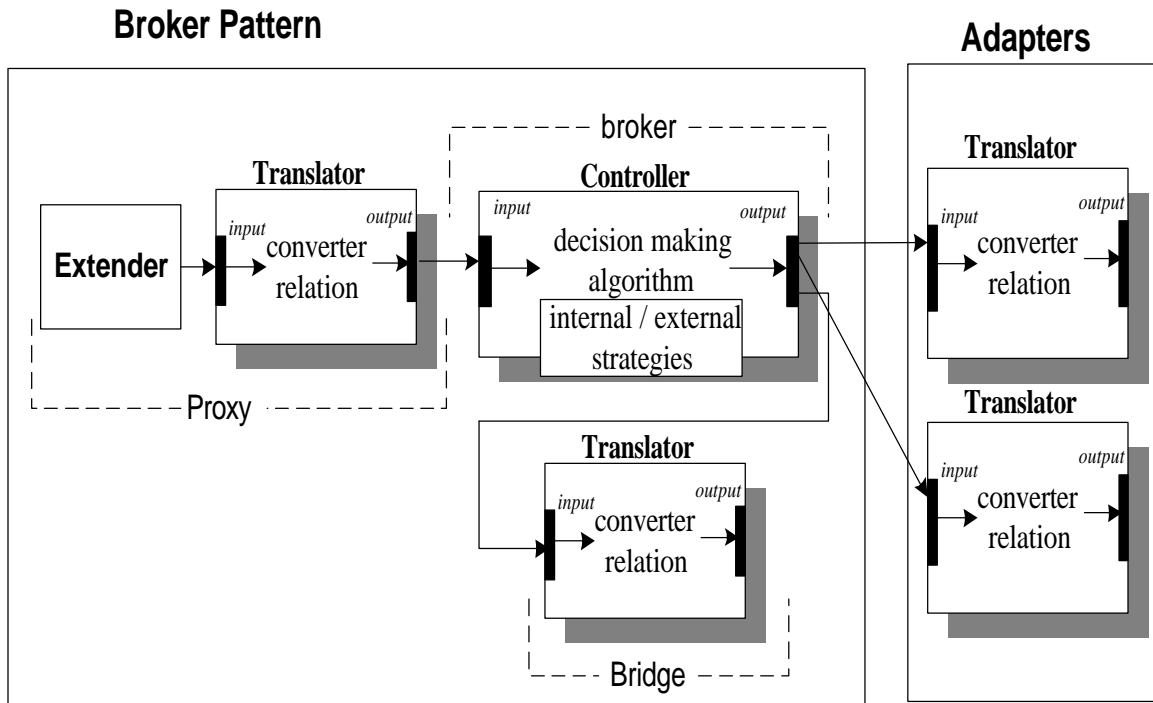


Figure 17: CORBA's Integration Architecture

5.2 DCOM

Distributed Component Object Model (DCOM) is similar to CORBA. It is Microsoft's solution to the interoperability problems of integrating components. DCOM can be applied to a Windows, Macintosh or Unix operating system environment [url4]. DCOM intercepts calls from clients and forwards requests to the corresponding component. DCOM's architecture consists of two Proxies whose functionalities are similar to those described in the Proxy pattern (Section 4.5.1) [BMRSS96, GHJV95]. The first proxy sits between clients and the DCOM architecture while the second is between the DCOM architecture and servers. The functionality of the Proxies depends upon the application. In general they perform security checks, data translation, and component synchronization. Figure 18 depicts the underlying integration architecture taxonomy.

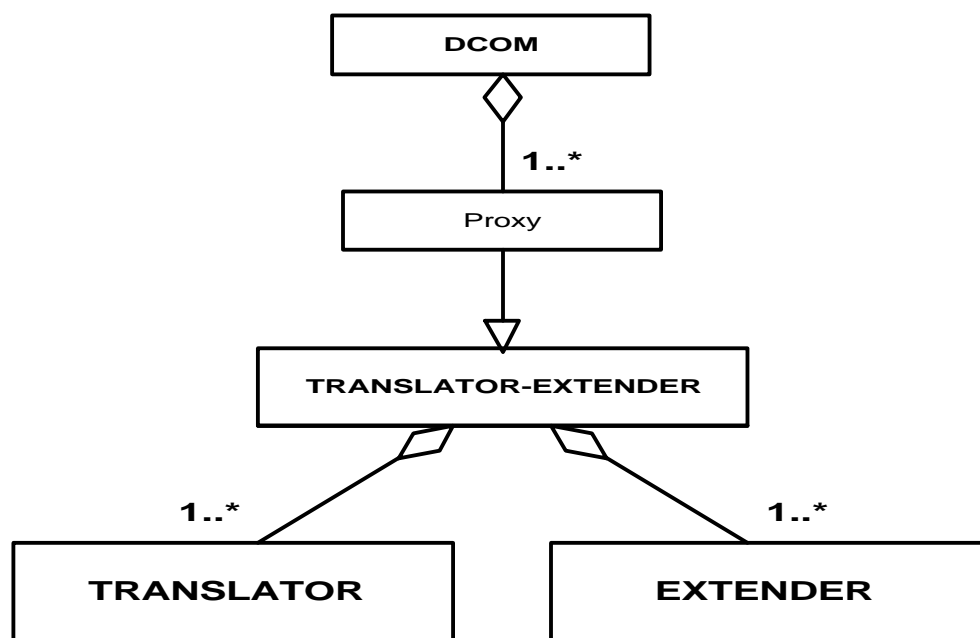


Figure 18: DCOM in the Integration Taxonomy

5.3 Java RMI

Java Remote Method Invocation (Java RMI) uses remote method invocation for remote object execution across a network. Java RMI is used by client code and calls methods on Java objects residing on remote machines (a server). RMI clients and servers can use basic methods as a Bridge (see Section 4.1.1) for communicating with existing and legacy systems [url5]. When a client wants to remotely invoke a method on a server, RMI downloads a stub that translates the invocation into a remote method call to the server. The stub's functionalities are similar to the Bridge pattern (Section 4.1.1) of Gamma et al. [GHJV95]. Implementing this bridge allows new clients with different data types to be easily added, creating a flexible system. On the server side, the call is received by the RMI system and connected to a "skeleton", which invokes the appropriate method based upon the input [url5]. The skeleton acts like a controller. In Figure 19 we can see that the underlying integration architecture of Java RMI contains the bridge

pattern and a type of controller.

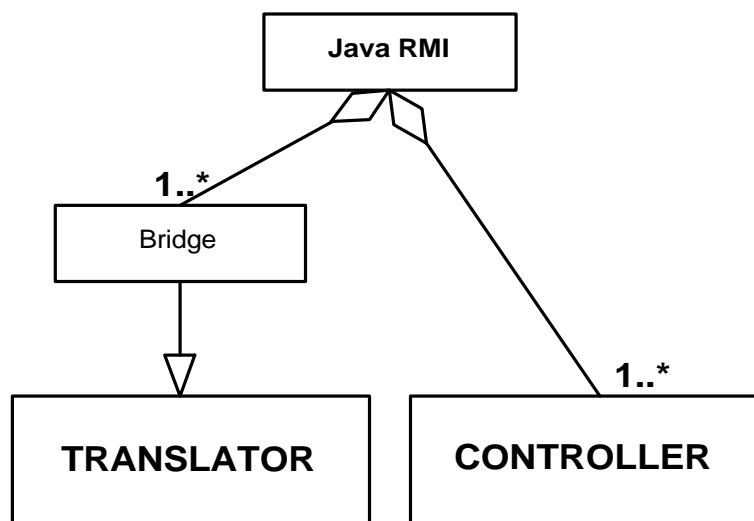


Figure 19: Java RMI in the Integration Taxonomy

5.4 JavaBeans

JavaBeans is mainly used for web page construction, visual application builders and graphical user interface (GUI) layout designers [url3]. It can be considered middleware because it allows new applications to be composed out of previously independent and stand-alone applications in the form of applets. JavaBeans was developed by Sun and is supported by vendors such as Apple, Borland and Oracle. JavaBeans is a class whose architecture is based on the components contained in the class. The JavaBeans architecture often contains several types of adapters whose core functionalities are similar to the Adapter pattern (Section 4.1.2). Example adapters are the event adapter, the demultiplexing adapter and the OKButtonAdaptor. Event adapters are key components of the Java event model [url3]. When additional behavior is required during event delivery, an intermediary event adapter class is defined that goes between an event source and the real event listener [url3]. The Java Bean's architecture uses Java

RMI as a mechanism to access the internet [url3]. JavaBeans uses the industry standard OMG CORBA distributed object model as its method of communication for talking to clients that use the Java IDL servers and other non-Java IDL servers. The architecture of JavaBeans is rather complex (Figure 20) and contains the adapter pattern, Java RMI and CORBA.

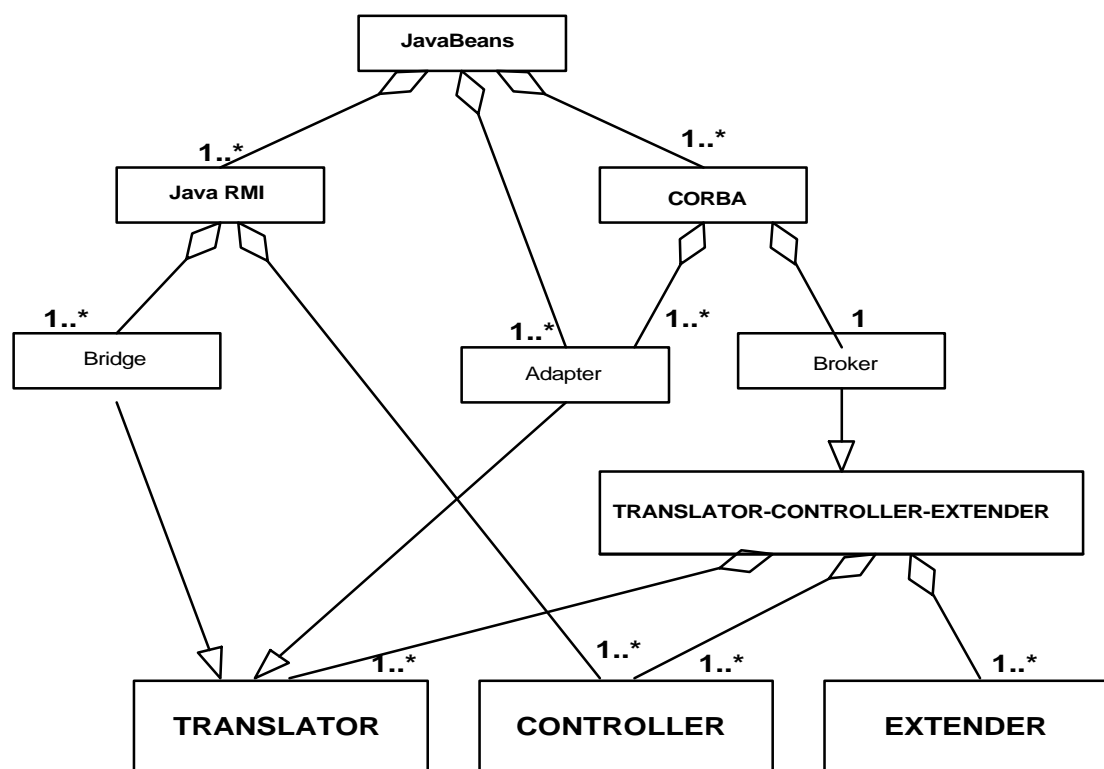


Figure 20: JavaBeans in the Integration Taxonomy

5.5 MQIntegrator

MQIntegrator is a middleware toolkit by IBM and New Era of Networks (NEON) that makes it easier to pass messages between applications [url1]. It is a real-time solution, which enables messages to be transmitted and processed as soon as they are sent and received. There are four components in the MQIntegrator: MQSeries, NEON Formatter, NEON Rules, and MQIntegrator daemon.

The MQSeries is the leading message-oriented-middleware product and can handle high value and high volume messages. It also provides assured once-only message delivery. MQSeries contains a queue manager that separates the communicating programs so that a sender can continue processing without waiting for a reply [url1]. The queue manager is like a controller because it identifies the destination of messages and places them in the corresponding queue of the servicing component or passes it to another queue manager. This controller is similar to the dynamically composed [Lea94] controller, whose invocation rules are based on incoming data (Section 4.2.5).

The NEON Formatter is a translator that restructures data in its queue to the form that the receiving application expects. The formatter has a pattern matching method whose rules are stored in a relational database [url1]. This Formatter is similar to the Filter pattern [Lea94], which is a stateless function that transforms data from one form to another (Section 4.1.3).

Once the data has been translated, the controller in the Neon Rules component uses a message routing procedure to decide the best route to the destination [url1]. This controller is similar to the rulebased controller [SGa97 GSP99, Butler98] because decisions in both of these controllers are based on rule matching.

The MQIntegrator controls the flow of data through out the MQIntegrator process. The MQIntegrator daemon processes the MQSeries's input to the Formatter and NEON Rules and decides which route the message should take and where it should be stored [url1].

We have been able to determine that the MQIntegrator architecture is based on *at*

least the following patterns: dynamically composed controller [Lea94], Filter [Lea94] and the rulebased controller [SGa97, Butler98] (Figure 21).

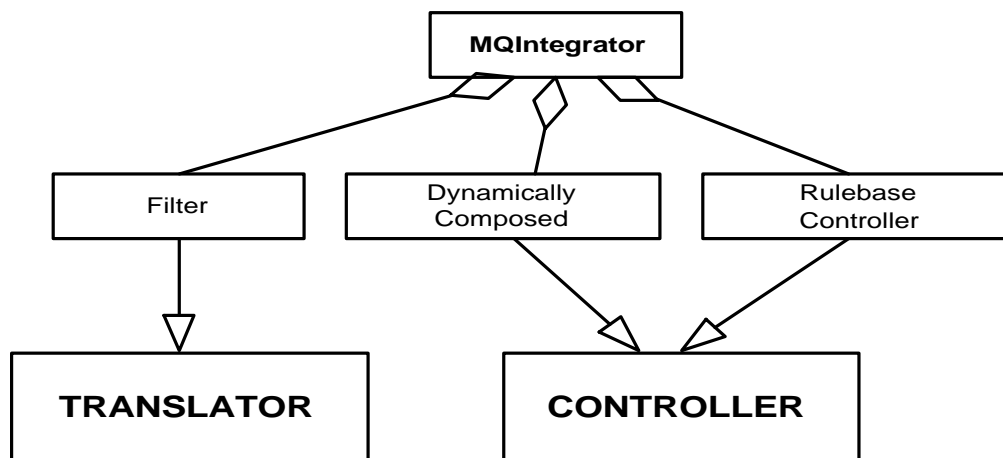


Figure 21: MQIntegrator in the Integration Taxonomy

5.6 Component Broker

The component broker is an IBM solution to interoperability problems that occur during Web based integration. The component broker enables the developer to build, run and manage a Web based system [McFall98]. The architecture of the component broker is similar to CORBA. It consists of an ORB and several application adapters. The ORB functionality is similar to the (see Section 4.7.1) Broker pattern found in [BMRSS96, Mularz94] while the adapters perform the same functionality as the adapter pattern in [GHJV95]. Figure 22 shows how the underlying integration architecture of the Component Broker fits into the integration taxonomy. We see that the Component Broker contains the adapter pattern and the broker pattern, and this embeds the functionalities of all three integration elements (translator, controller and extender).

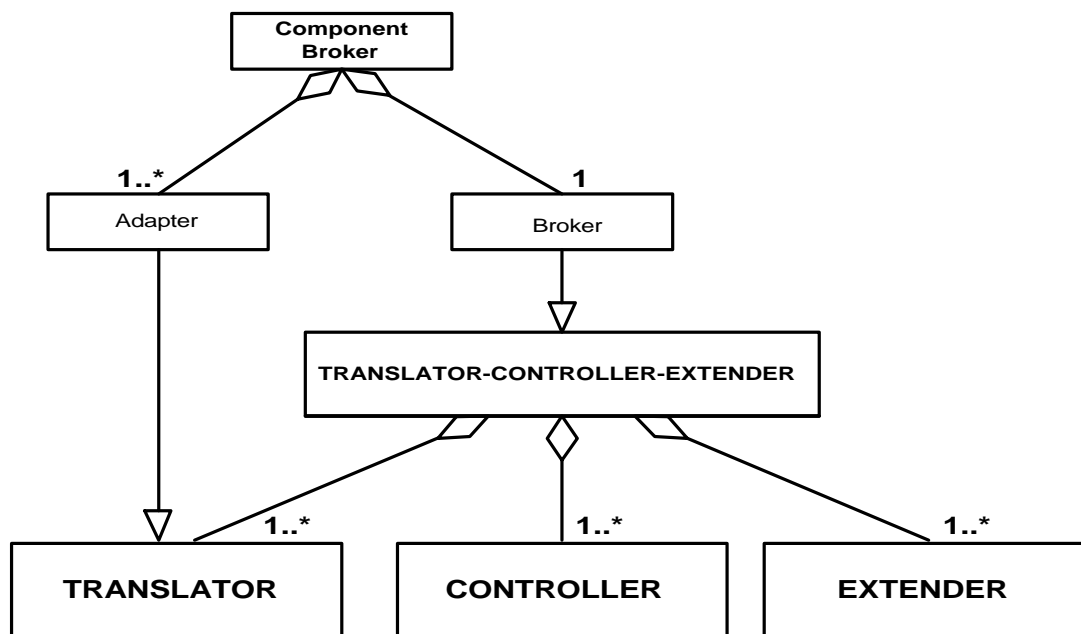


Figure 22: Component Broker in the Integration Taxonomy

5.7 Polyolith

Polyolith is a middleware solution developed at the University of Maryland [Purtilo94]. It was built to allow applications to communicate across process boundaries using messages made up of arbitrarily complex type structures. The Polyolith architecture consists of two important components that directly relate to integration: the software bus and the packaging system. The clients pass requests to the packaging system, which acts as the system controller. Depending on the request, the corresponding software bus is used which connects the packaging system to a component. It also translates the data into a form that the server component expects. This is similar to the functionalities of the Mediator pattern presented by Gamma et al. [GHJV95] (see Section 4.4.1). The mediator pattern allows the developer to integrate a system without having to change the source code of either the client or the server. In the integration taxonomy (Figure 23) we see that Polyolith contains the functionalities of the mediator pattern.

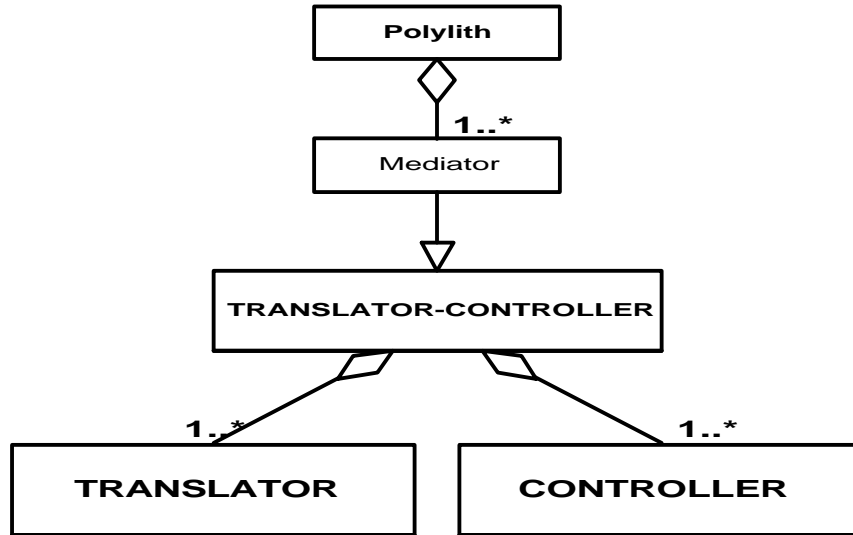


Figure 23: Polyolith in the Integration Taxonomy

CHAPTER VI
CONCLUSION AND FUTURE WORK

6. Conclusion and Future Work

The evolving global economy is driving companies towards component based software engineering. Due to the lack of interoperability among existing systems, this type of development is not an easy task. The need for reliable and quick solutions to interoperability problems has produced an explosion in the number of integration solutions and middleware products available. However, these solutions are at a lower level of abstraction, either close to or at the implementation level of development. Because these solutions are chosen at this stage, many difficulties can arise. Due to the lack of architectural information concerning interoperability problems, developers may not understand the exact cause of the interoperability problems or which solutions best resolve the problems. Furthermore, developers may choose a strategy that contains more functionality or is more complex than necessary. This research addresses these issues.

In this thesis, the analysis of current integration solutions at the architectural level reveals three architecture integration elements: Translators, Controllers and Extenders. The translator and controller are indivisible connector models that form the building blocks of integration functionality, while the extender's functionality enhances the services provided by translators and/or controllers. These elements and their combinations form the foundation of the integration taxonomy.

Further research may reveal that the extender's functionalities can be further divided to form several indivisible integration elements. Further research is also needed in defining the characteristics of a system at the architectural level. This will allow developers to search for integration mismatches at the architectural level. Using the

architectural elements and the taxonomy, the developer can then find a suitable implementation solution.

The taxonomy uncovers a common denominator between the integration elements, the integration strategies, and middleware products by revealing their core functionalities. This connection between architectural and implementable solutions allows for integration to become a design decision. Since the interoperability solutions are chosen at the design level, the developer is able to document the reasoning behind the choice of solutions and can later trace these decisions as the system needs to evolve.

Another benefit of the integration taxonomy is that it can be used to distinguish integration patterns from other types of patterns, which narrows the number of patterns the developer needs to consider as solutions during integration. The taxonomy can also be used to compare the functionalities of the integration strategies being considered as solutions to the interoperability problems, which enables the developer to choose the best solution. If a suitable solution is not found, the integration taxonomy provides integration architecture information that can be used to build custom interoperability solutions.

In the future, adding additional potential interoperability solutions (integration strategies and middleware solutions) will enhance the usefulness of the integration taxonomy and prevent developers from creating customized solutions when they have already been developed. These solutions functionalities must be analyzed at the architectural level before they can be considered for the integration taxonomy.

CHAPTER VII
REFERENCES

7. References

- [AbdAllah96] A. Abd-Allah. Composing heterogeneous software architectures. Ph.D. Dissertation, Dep. of CS, USC, August 1996.
- [AAG95] G. Abowd, R. Allen and D. Garlan. Formalizing Style to Understand Description of Software Architecture. *ACM TOSEM*, 4(4):319-364, 1995.
- [AG97] R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM TOSEM*, 1997.
- [BCTW96] D. Barret, L. Clarke, P. Tar, and A. Wize. An event-based software integration framework. Technical Report 95-048 (revised 1/96), Laboratory for Advances Software Engineering Research, Computer Science Dept., Univ. of Massachusetts, 1996.
- [BMRSS96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons Ltd. New York, NY, 1996.
- [BPEA99] B. Boehm, D. Port, A. Egyed, and M. Abi-Antoun. The MBASE life cycle architecture milestone package. *1st Working International Conference on Software Architecture*. February 1999.
- [BS95] M. Brodie and M. Stonebraker. *Migrating Legacy Systems: Gateways, Interfaces & the Incremental Approach*. Morgan Kaufmann Publishers, Inc. San Francisco, CA, 1995.
- [Butler98] P. R. (Butler) Stiger. An assessment of architectural styles and integration components. M.S. Thesis, Dept. Mathematical and Computer Sciences, University of Tulsa, 1998.
- [Charles99] J. Charles. Middleware moves to the forefront. *Computer*. 17-19, May 1999.
- [Cole97] D. Cole. Integration patterns for the component web. M.S. Thesis, Dept. MCS, University of Tulsa, 1997.
- [Comer95] D. Comer. *Internetworking with TCP/IP Vol. 1: Principles, Protocols, and Architecture*, 3rd edition, Prentice Hall, 1995.
- [Chen95] C. Chen. Integrating existing event-based distributed applications, Xerox Corporation, 1995.
- [Deline99] R. DeLine. Techniques to resolve packaging mismatch. *ICSE 99*. 1999.
- [DKRS91] R. Duke, P. King, G. Rose, and G. Smith. The Object-Z specification

language: version 1. Software Verification Research Center, Department of Computer Science, The University of Queensland, Technical. Report 91-1 (1991).

- [DMT] E. Dashofy, N. Medvidovic and R. Taylor. Using off-the-shelf middleware to implement connectors in distributed software architectures. *21st International Conference on Software Engineering*, Los Angeles, CA, May 1999.
- [GAO95] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch or why it is so hard to build systems out of existing parts. *Proceedings of the 17th International Conference on Software Engineering*, April 1995.
- [Gacek97] C. Gacek. Detecting architectural mismatches during systems composition, TR USC/CSE-97-TR-506, USC, Center for Software Engineering, 1997.
- [GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Company, Reading MA 1995.
- [GSP99] R. Gamble, P. R. (Butler) Stiger, R. Plant. Rule-based systems formalized within a software architectural style. *Journal of Knowledge Based Systems*, forthcoming 1999.
- [HGKS99] K. Hasler, R. Gamble, K. Frasier, and P. (Butler) Stiger. Exploiting inheritance in modeling architecture abstractions. *1st Working IFIP Conference on Software Architecture*, February 1999.
- [ITU96] International Telecommunication Union, Principles for a Telecommunications management network. ITU-T M.3010, 1996.
- [KG98] R. Keshav and R. Gamble. Towards a Taxonomy of Architecture Integration Strategies. *3rd International Software Architecture Workshop*. November 1998.
- [KG99] A. Kelkar and R. Gamble. Understanding the architecture characteristics behind middleware choices. TR UTULSA-MCS-99-20. Under review 1999.
- [Lea94] D. Lea. Design patterns for avionics control systems. SUNY Oswego & NY CASE Center, DSSA Adage Project ADAGE-OSW, 1994.
- [McFall98] C. McFall. An object infrastructure for internet middleware: IBM on Component Broker. *IEEE Internet Computing*. March/April 1998.
- [Mularz94] D. Mularz. Pattern-based integration architectures. *PloP*, 1994.

- [OH97] R. Orfali and D. Harkey. *Client/Server Programming with Java and CORBA*. John Wiley and Sons, New York, 1996.
- [PKG98] J. Payton, R. Keshav, and R. Gamble. System development using the Integrating Component Architectures Process. *ICSE 99 Workshops on Successful Development of COTS Based Systems*. 1999.
- [PW92] D. Perry and A. Wolf. Foundations for the study of software architecture. *SIGSOFT Software Engineering Notes*, 17(4): 40-52, October 1992.
- [Purtilo94] J. Purtilo. The Polyolith software bus. *ACM Transactions on Programming Languages and Systems*, Jan. 1994.
- [Rational99] Rational Software. <http://www.rational.com>.
- [SC97] Mary Shaw and Paul Clements. A field guide to boxology: preliminary classification of architectural styles for software systems. Proc. COMPSAC97 and *1st Int'l Computer Software and Applications Conference*, August 1997, pp. 6-13.
- [SG96] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.
- [SG99] M. Shaw and M. Gentleman. Integration & interoperability working group report. *1st Working IFIP Conference on Software Architecture*, 1999.
- [SGa97] P. R. (Butler) Stiger and R.F. Gamble. Blackboard systems formalized within a software architectural style. *Int'l conference on Systems, Man, Cybernetics*, October 1997.
- [Sh95] Mary Shaw. Some patterns for software architectures. *2nd Annual Conference on the Pattern Language of Programmers*. May 1995.
- [Sitaraman97] R. Sitaraman. Integration of software systems at an abstract architectural level. M.S. Thesis, Department of Mathematical and Computer Sciences, University of Tulsa, 1997.
- [Spivey88] J. Spivey. *Introducing Z: A Specification Language and its Formal Semantics*. Cambridge Univ. Press 1988.
- [Sridhran97] P. Sridhran. *Advanced Java Networking*. Prentice Hall, NJ. 1997.
- [url1] MQIntegrator: A Technical Overview. <http://www.software.ibm.com/ts/mqseries/integrator/mqintegrator/whitepapers/eai-tech/>
- [url2] CORBA. <http://pent21.infosys.tuwien.ac.at/Research/Corba/OMG/arch2.html>

- [url3] JavaBeans. Sun Microsystems. <http://java.sun.com/beans>. July 24, 1997.
- [url4] Microsoft Windows NT server DCOM technical overview white paper. http://msdn.microsoft.com/isapi/msdnlib.idc?theURL=/library/techart/msdn_comprr.htm
- [url5] Java Remote Method Invocation - Distributed Computing For Java. Available at <http://www.javasoft.com/marketing/collateral/javarmi.html>
- [Vinoski96] S. Vinoski. CORBA: Integrating diverse application within distributed heterogeneous environment. *IEEE Communications Magazine*, February 1997.