Testing Component-based Software Systems

Andrea Polini

Scuola Superiore di Studi Universitari e Perfezionamento "Sant'Anna"
Settore di Scienze Sperimentali
Diploma di Perfezionamento in Ingegneria Informatica

Testing Component-Based Software Systems

Ph.D. Dissertation of:
Andrea Polini

Advisor Board:

Prof. Paolo Ancilotti

Ing. Antonia Bertolino

Ing. Giuseppe Lipari

Dott.ssa Emilia Peciola

November 2004

SCUOLA SUPERIORE DI STUDI UNIVERSITARI E PERFEZIONAMENTO “SANT’ANNA”
Settore di Scienze Sperimentali
Piazza Martiri della Libertà, 33   I 56127 — Pisa

Contents

List of Figures

Preface

This thesis presents the main subject and results of the research that I carried out during this three years and a half. At the beginning of this period I was rather unacquainted with “software components”. It has been a piece of a quite hard work but I am happy of having taken the decision of doing it.

This thesis is organized in four successive parts. In the first part I discuss the tools available for modeling and implementing a complex software system as an assembly of software components. In Part II I introduce concepts related to system verification, with particular emphasis to functional testing and performance evaluation of component-based software systems. Part III discusses the three main proposal for component-based software system evaluation that I developed during my thesis in cooperation with other authors. I conclude the thesis highlighting some conclusions and possibility for future work.

At the end of this experience I have to thank many people and for many different reasons. Following a sort of chronological order, I want to thank Prof. Piero Maestrini for having encouraged me in taking part to the PhD program at Scuola Sant'Anna. A great thank to Ericsson Lab Italy for the financial support. Thanks to the Scuola Superiore Sant'Anna and in particular to my advisors, Prof. Paolo Ancilotti, Ing. Antonia Bertolino, Ing. Giuseppe Lipari and Dott.ssa Emilia Peciola. A great thanks also to Prof. Anthony Finkelstein and Prof. Wolfgang Emmerich for having hosted me at Computer Science Department of the University College of London.

A particular thanks to Antonella for the uncommon humanity and for the passion that she puts in her work. Thanks to the fantastic guys and girls that I met at ISTI-CNR. Thanks to Eda, Francesca B, Francesca L, Francesca M, Enrico, Guglielmo, Alberto, Daniela, Sara, Matteo and many others. Thanks to all the friends that I met at UCL during my stay there. In particular to Giovanni that made London and working there more interesting, dangerous and funny. Thanks to the many friends from L'Aquila and in particular to Henry which has always in his rucksack interesting scientific problems to solve and discuss.

At the end a special thanks to my wonderful family to my parents Luca and Regina, to my nephews Paolo and Mattia, to my brothers, Daniele, Davide, Massimo and to

his wife Romina, because is always a pleasure to have the time to go back home and it is a shame not having so much time. Finally I great thanks to Viviana for the happiness that she uses to discover the day.

1. Introduction

The latest years can be certainly characterized by the increasing pervasiveness of information processing systems. Many new application fields have been explored and many new software systems have been developed. On the one end, this testifies that the trust on software has generally grown, to the effect that it is more and more used in risky activities, of which the on-line banks are perhaps one of the most evident examples to the general public. On the other end, this also makes it mandatory to enhance the "-ilities" of the produced software, while assuring a high dependability, or otherwise the consequences can be catastrophic. This trend does not give any sign to be going to finish yet, and as a result the complexity of software systems is continuously rising. At the same time the software developers, to stay competitive, need to cope with the constant reduction of the time-to-market.

The answer to these challenges is being sought on the potential to obtain complex systems by composing prefabricated and adequate pieces of software called "components". Following in this direction the example provided by other engineering disciplines, the simple idea underneath is that building complex systems by assembling already produced subsystems (components) should be faster and easier than rebuilding them from scratch. At the same time, it is supposed that reusing subsystems, whose qualities have been verified as part of earlier "successful" systems, should grant a higher reliability. However, some laboratory experiments [90] and even catastrophic events [109] have soon warned that composing components is not an easy task and much research is necessary to enable this vision. As a consequence a new research branch inside the software engineering area has been established, with the aim of studying and developing methodologies and technologies for the dependable composition of components. This branch is generally referred to as the Component Based Software Engineering (CBSE). The raising interest from both academy and industry, and then the raising importance of this new discipline, is testified by the spreading of devoted events (e.g. conferences specifically oriented to discuss different component related topics, such as [70, 80, 71, 52]),journal (e.g., [19, 69]) and books (e.g. [160, 68, 48].

CBSE has to face many challenges. In [67] the author provides a quite long list of research topics, related to CBSE, that need to be further explored. With a particular reference to the contents that will be discussed in the following of this thesis, major researches are certainly necessary in the area of:

1. Modeling languages: it is necessary to develop languages to express models of the

software systems at an high level of abstraction, ignoring at that level unuseful technical details. In particular such languages should permit to describe the architecture of the system in term of coarse grained component and of their required features.

2. Technologies: it is necessary to develop suitable technologies to make easier the integration and communication of software components. Technologies should provide high level services that permit to easily bind together components and make them to cooperate without annoying the software assembler with networking and non business logic details.

3. Development Process: it is necessary to better understand which are the steps that can lead to the development of a system starting from components. It is clear that we need a highly iterative process that permits to consider features of reused elements starting from the first phases of the development. This process will foresee phases specifically relative to CB development, such as a provisioning phase that should permit the identification of suitable implementation for the components in the architecture.

4. Tools: it is necessary to develop tools that assist the developer during all the phases of the development process. With reference to testing it is particularly important to implement tools that make easier the test of externally acquired components. Testing could, in fact, be fruitfully used for the evaluation and final choice of an external component, and this step could involve many different components.

5. Composition Predictability: two different points of view can be considered for this requirement. The first involves to infer interesting properties of a system composed from components, starting from the known properties of the composing components. The second point of view takes the orthogonal starting point. In particular it consists of inferring system properties starting from the study of the logical architecture of the global system. A particularly relevant property that will be further discussed in the thesis is performance.

All the elements in the list are strongly interrelated with the material presented in this thesis. For the last two points novel solutions are proposed, instead the first two points in the list constitute the necessary background in which the solutions proposed find their justification. Finally, in this thesis I do not discuss in detail any specific development process even though some simple assumption on the process will emerge from the discussion. The hypothesis is that the solution proposed will not depend on a specific process, since whatever is the process adopted the data necessary to apply the approaches should be always available.

This thesis is structured in three parts. Part I serves to give an introductory overview of the particular research field in which this thesis has been conceived and in which the solutions proposed should find opportunities for being applied. It comprises two chapters. Chapter 2 is on modeling.

The modeling phase of complex software system received a lot of interest in the last ten years and useful instruments to address the problem have been provided mainly by three research field, such as:

1. Software Architecture
2. Unified Modeling Language
3. Model Driven Development

These research domains are obviously strongly interrelated and study have been conducted to merge the proposed solutions.

Chapter 3 is instead devoted to the discussion of the technological aspects of CBSE. Therefore I present in it the two most important pieces in the CBSE picture, such as middleware and CB models.

During my PhD I mainly carried out researches to find effective solutions for functional and non functional evaluation of component based systems. Part II in this sense wants to give an overview of the solutions proposed by other research groups for problems similar or related to those that I studied during my PhD. In particular Chapter 4 gives an overview on testing. The presentation will provide basic notions about testing permitting a better understanding of the following chapters. In Chapter 5, then, I discuss the solutions proposed for testing execution when the considered System Under Test (SUT) is a software component or a subsystem composed assembling components. In that chapter I provide a classification of the other solutions proposed in this area and I highlight how the focus is generally oriented on providing techniques to transfer test cases, established by the component developer using its specifications, to the final component user (or system assembler). Finally I discuss the main drawbacks behind the adoption of this kind of solutions and I explain why it is useful to have a framework that makes the testing of components easier by using test cases defined by the final component user starting from the specifications that he/she has developed.

In the following chapter of Part II I introduce the problem of evaluating the performance of a complex software system from the first phases of the development. The solutions proposed by other authors are classified as belonging to two big categories. The first intends to use analytical approaches that are based on inferences derived from the definition of a model of the system. Generally these solutions try to attach to the model representing the system, that usually consists of diagrams defined using the Unified Modeling Language (UML - see Chapter 2), information concerning performance characteristics. After this step the general solution is to apply formalisms based on sound and precise mathematics that provide as result an approximation of the actual system performance. The second proposal instead suggests the use of testing to empirically derive an evaluation of the performance for the system. The idea of testing could seem counterintuitive if we consider that we want to provide an evaluation of the performance starting from the first phases of the development and being testing a mechanisms that requires the real execution of a system. For this reason solutions in this area require the definition of some kind of prototype for the system under development that should approximate as much as possible the real system at least for what concerns "performance behavior".

Finally in Part III I discuss three main research studies that I carried out in the area of component evaluation. In particular in Chapter 7 I describe a framework, that I developed in collaboration with Antonia Bertolino, to facilitate the execution of test cases against components acquired externally. Objective of the framework is to provide a useful mechanism to the system assembler for defining test cases that could be successively used for testing software components and subsystems. In particular using the framework the systems assembler can codify, starting from the first phases of the development, test cases derived from the specific requirements for a components foreseen by the architecture of the system. When a real implementation for a component in the architecture is identified, the test suite will be executed against the identified component using a dynamic adaptation step. In that manner the framework permits to separate the codification phase, made once and for all, from the adaptation phase, with a real benefit in terms of effort saving and reducing the risk of making mistakes during the tests codification phase.

In Chapter 8 I discuss an approach for the derivation of test cases for system that will be developed assembling components. The work presented is still an ongoing research that I am exploring in collaboration with Antonia Bertolino, Paola Inverardi and Henry Muccini. In the chapter, starting from a "critic" to model based testing when the system to be tested will be derived assembling components externally acquired, I propose a novel approach that tries to derive a behavioral model of the system from the execution of test cases derived using an operational profile. The derivation of the model requires the introduction of tracing mechanisms, that are discusses in the chapter, and the study of suitable synthesizing algorithms that will enable the derivation of a behavioral model from a set of execution traces. As final steps the approach proposes to use the derived model to apply techniques, such as model checking, that permit to infer properties on the system under study.

The last chapter of Part III (Chapter 9) describes a novel approach for empirical evaluation of CB systems, that I developed and studied in collaboration with Giovanni Denaro and Wolfgang Emmerich. The approach wants to provide a methodology for the early evaluation of system performance from the first phases of the development. It has been developed considering as subject of study distributed systems that use complex middleware. Since middleware is difficult to model using analytical approaches and at the same time it strongly influences system performance, we think that these approaches cannot provide trustable prediction of system performance. The solution, that I describe, provides a way for deriving a prototype of the system, and use test cases, selected following an operational profile to have an empirical evaluation of the systems performance. I report also some initial experiments that seem to indicate the real effectiveness of the proposed approach for performance prediction.

The thesis ends with Chapter 10 in which I summarize the concepts presented and the solutions proposed. A brief overview of possible directions for further researches is also discussed.

Part I.

Developing Component Based Software Systems

This part depicts the general landscape in which the researches reported in this thesis should be considered. In particular I introduce here the instruments that constitute the theoretical and practical background for the conducted studies. Therefore I discuss:

- In Chapter 2 languages for the definition of complex software systems, focusing the attention on architectural concerns;
- In Chapter 3 the technologies that support and make practically tractable the implementation of component based software systems.

2. Modeling Component-based Software Systems

In the last years software industries have witnessed a growing rate of failures in the development of software system projects. This trend emerged mainly as a consequence of the current need of producing more and more complex software systems, often within strict release deadline, and once again testifies that software engineering is not a well established and mature discipline, yet. Lack of methodologies and techniques for raising the logical level at which it is possible to reason and make inferences about software systems can bring to failures. In fact, as the software system complexity increases, the importance of the overall system structure, or model of the system, become a more significant question than the choice of particular algorithms and data structures for the computation. Objective of modeling is, in fact, to provide an abstraction of a physical system allowing engineers to reason about that system by ignoring extraneous details while focusing on relevant ones [57].

Models are used by all engineering disciplines to capture significant features/information on the system under construction, having an abstract but correct representation of the system. The information captured in the model will be necessary to perform all the following steps towards the obtainment of the real implementation. No one could imagine the construction of a bridge or of a car without first the study and development of specialized models. Nowadays, software artifacts are probably the most complex systems that have ever been constructed, therefore, as Bran Selic recognizes [149], the potential benefits on the use of models are significantly greater in software than in any other engineering discipline. However, in spite of the assessed software system inherent complexity, and of the past failures in software production, the use of clear methodologies and languages for the definition of models is, in software industries, relatively scarce. In a recent interview to Grady Booch and Bran Selic about the use of modeling language emerged that, only the ten percent of the projects developed by industries are carried on using precise models [154]. Industries generally prefer, in fact, to entrust their fortune on technical advances, such as advanced development environments, or process improvement, rather than let the programmers spending time making software models (as further confirmation, programmers value are often assessed as line of code per day). In fact, technical advances, even though they can improve efficiency at the lowest level of system development, do not have beneficial influences at the more abstract levels. Therefore implementation phases

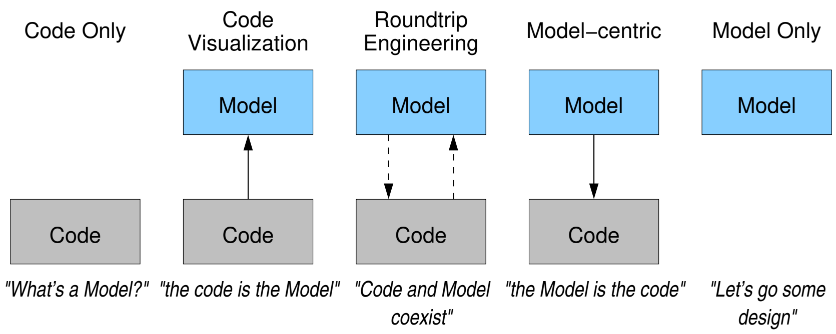

Figure 2.1.: The modeling spectrum (from [57])

generally begin too early, putting too much effort on the coding stage. A funny sentence, that sounds paradoxical but indeed hides some deep truth, and that should be much more considered by software developers, with reference to the coding phase, says: "*later you start before you finish*", highlighting that a longer modeling phase could guarantee a shorter coding phase and then a shorter release time.

Figure 2.1 outlines the possible relations between code and software model, in what can be called a "modeling spectrum". Each category identifies a particular use of models in aiding software developers to create running applications. On the left end side the code is the only artifact that will be built during the development process; informal models will be directly expressed in the programming language used. The second category represented, identifies the case in which models are derived from the already developed code providing a concrete visualization of the developed system. This should make easier the understanding of the system structure and the manipulation of it, but obviously does not have any influences on the derivation of the code. The category in the middle, called round-trip engineering, represents the case in which a strict relation among code and model permits to directly modify one of them having a correspondent modification on the other. The model in this case is, in fact, defined at a very low level having the same level of detail of the source code. However differently from the previous case, in this situation the model is also used to directly bring modifications to the code. In the model-centric approach (in Figure 2.1 the second from the right end side), the code is derived starting from an abstract model that synthesizes the important features of the system, disregarding unuseful details. From the abstract model, through several phases in which further details are added, the final code is defined. Finally on the right end side the model-only category represent the case in which models are used mainly to communicate idea and to better understand a proposed solution. In this case, there is no strict relation with a real implementation.

In our discussion we focus the attention on those tools that are oriented to the definition of complex systems using a model-centric view. As first example of this kind of tool we can list methods that were mainly based on the use of box and lines diagrams. Those kinds of tool mainly emerged as sporadic initiatives inside different organizations, each one using different symbols, and without a clear semantic associated to each element. Obviously the produced models resulted rather rough. The presence of ambiguities caused lot of problem towards the obtainment of a real system consistent with the model. Instead, what we need is to develop methodologies and techniques on which a wider agreement can be reached, and permitting to express models with an unambiguous meaning. A mature engineering discipline needs, in fact, to express inexpensive models from which it should be possible:

- to analyze and derive system properties
- to maintain the system
- to evolve the system
- to understand the system structure and functioning
- etc...

The research in this area is not completed yet, but certainly important results have been marked. The instruments that I present in this chapter have earned much attention on the area of modeling for component based software systems, and even though they have been derived to address the modeling phase at different levels of abstraction they are strongly interrelated, and, as we report in the following, some studies have been conducted trying to merge them.

2.1. Modeling Views

Capturing all the aspects of a complex system in a single diagram is practically impossible. As a consequence an important concept strongly related to that of modeling, is that of "model views". The idea has been borrowed from all the other engineering disciplines in which it is normal to use different description to illustrate different aspects of the project. Perry and Wolf [139], in that is considered one the seminal paper in Software Architecture investigation, with reference to the civil engineering note that the construction of a building foresee the definition of different views such as elevation and floor plans that provide the exterior views and the horizontal views respectively. From this analogy becomes of basic importance to identify useful point of view for the meaningful description of software systems.

Different aspects can be considered in the organization of a model views, and which view should be included is not unambiguous. Obviously the definition of a "model views" will reflect the aspects that are considered more important by the developer of the model (e.g. major focus on functional aspects instead of nonfunctional aspects). Perry and Wolf [139] proposed in their work a first list composed of three different views, such as **processing view**, **data view** and **connections view**. Successively another views model was proposed by Soni, Nord and Hofmeister [156]. In their work the authors propose a model based on four view respectively called **conceptual architecture**, **module interconnection architecture**, **execution architecture**, and **code architecture**. However the model that have certainly had more fortune until now, since is often used with the Unified Modeling Language [13], has been defined by Kruchten [105] and is generally know as the "4+1 View model". The model views proposed by Kruchten is constituted by:

1. **logical view**, shows how the system's functionality is provided. It provides a representation of the inside of the system in terms of static structure and dynamic behavior, providing information about the relation that must be hold among the elements in the system.

2. **process view**, deals with properties such as performance and system availability. This aspects, which are nonfunctional properties of the system, allows for efficient resource usage, parallel execution and the handling of asynchronous events from the environment. Besides dividing the system into concurrently executing threads of control, this view must also deal with the communication and synchronization of these threads.

3. **development view**, describes the software's static organization in terms of software modules in the software-development environment. The development view takes into account internal requirements related to ease of development, software management, reuse or commonality, and constraints imposed by the tool-set of the programming language. This view also supports the allocation of requirements and work to teams, and support cost evaluation, planning, monitoring of project progress, and reasoning about reuse, portability, and security.

4. **physical view** describes the mapping of the software onto the hardware and reflects its distributed aspect. It takes into account nonfunctional requirements such as system availability, reliability, performance, and scalability. In this view the various elements defined in the other view must be mapped onto the various hardware processing node.

4+1. **use-cases view**, describes the functionality the system should provide, as perceived by the external actors. It constitutes a sort of glue among the other views (from this the number 4+1) driving their development. This vie can be also used to validate the system and to verify the functioning of the system by testing

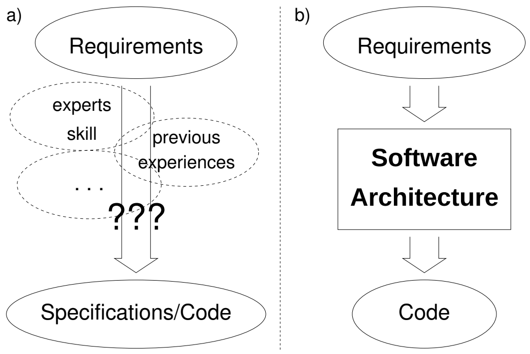

Figure 2.2.: From Requirements to Code

the use-case view with the customer ("Is this what the customer wants?")and against its real implementation ("Does the system work as specified?").

It is worth to note that the existence of separate views of the same system, raise problems of consistency among the different description that must be suitably addressed. In other word it is of basic importance that the information that are defined in one view do not contrast with choices described in another view. In [131] an interesting method for view consistency checking, based on model checking, is proposed as well as other possible options are discussed.

2.2. Software Architecture (SA)

Software Architecture emerged in the last decade as one of the most promising instrument to partially fill the gap between *requirements* and *specification/code* (see Figure 2.2). The traditional answer to this transformation was that something "magic" happens producing software from requirements. This magic is mainly related to the work carried out by experts on the specific field of development, that produce specifications from requirements and final system from the requirements, and it is the result of the high skill that they developed during the time. As consequence the process proceeds in an ad hoc and unpredictable way and it is almost impossible to communicate to third parties the choices taken during the process. The disadvantages of such development process are quite obvious, in particular with reference to the production of complex systems, and are related to the difficulties in carrying out the activities that we listed at the beginning of this chapter, such as early system analysis, instead enabled by the development and use of models and common notations.

Today Software Architecture is becoming an important subfield of Software Engineering and many researchers are currently working on this topic, at the same time important industries have started to introduce Software Architecture as one important step towards the development of the final system implementation [113, 101, 28]. However, in spite of these efforts, a general agreement on a unambiguous and precise definition of software architectures still has to emerge. In general we can say that the architectural design of a software system is concerned with its gross structure and the ways in which that structure leads to the satisfaction of key system properties [128]. In particular structural issues of interest for the SA description of a system include the composition of components, the definition of the global control structures, the definition of the protocols used for the communication, synchronization and data access, the assignment of functionality to design elements, the physical distribution of the component in the architecture, the scaling and performance feature of the system, the dimension of evolution of the system and finally the selection among different design alternatives [91]. Therefore from the architectural definition of a system it must be possible to identify the three basic elements that characterize a SA, such as **components**, the elements in which the logical computation is "located", the **connectors**, the elements that mediate the interactions among the components, and the **properties**, such as pre/post conditions, signatures, and RT specs. This information will be useful for construction and analysis phases.

Abstracting away from implementation details a good architecture description makes a system intellectually tractable and, as observed in [88], it plays a basic role in at least six aspects of software development:

1. *Understanding*: raising the level of abstraction at which the system is described, and disregarding unimportant details, Software Architecture provides a useful mean to describe the software system in a more understandable way. Facilitating also communication between teams.

2. *Reuse*: Reusing piece of code is today a central target in order to reduce the time to market in the development of complex system (we will extend the discussion on this topic in the following sections). In this area Software Architecture can play a central role grouping at a high level of abstraction the necessary functionalities and at the same time identifying the components that should provide such functionalities. Toward this objective the idea of framework and architectural design patterns can give an important contribution [86, 147, 59].

3. *Construction*: The architectural description provides a partial blueprint to the developers. In it, the different components and the relations among them are showed. Important objective for the developers is to follow the guidelines provided by the SA obtaining then an implementation that is consistent with the architectural description.

4. *Analysis*: Architectural description provide new opportunities for software analysis. In particular it can enable the identification of system lack at an early stage of the development reducing the cost spent to solve identified problem. Many different kind of analysis are possible such as:
   - system consistency checking
   - conformance to constraints imposed by an architectural style
   - dependence analysis
   - conformance to quality attributes
   - test derivation
5. *Evolution*: Software architecture can expose the dimensions along which a system is expected to evolve. By making explicit the "load/bearing walls" of a system, system maintainers can better understand the ramifications of changes, and thereby more accurately estimate costs of modifications.
6. *Management*: Having an intellectually tractable description of the system simplifies the action of developer that are responsible of system maintenance. Obviously it is important at this stage that software developer really maintain consistency with the architectural description.

These aspects mainly correspond to those advantages promised by the definition of a good model that we listed at the beginning of this chapter and testify that SA descriptions lead to the definition of good software models.

2.2.1. Architectural Description Languages (ADL)

So far we mainly discussed the benefits that a software engineer can gain from an architectural description of the system under construction. We also discussed the main motivation that brought to the success of this new discipline. However it is possible to take advantage from a description of a system at the architectural level only if the tools used to describe that system permit some kind of analysis and inference. This generally means that it is necessary to develop formal languages for SA description. In that manner it will be possible to unambiguously define specification of intended system architecture, avoiding the problem of different interpretations, from different stake-holders, of the same architectural description. As noticed in [91] some important properties should be embodied by a good language for architectural definition in order to obtain a valuable description. They are:

- **Composition**
  "*It should be possible to describe a system as a composition of independent computational elements and intercommunication elements*". As a consequence it

should be possible for the system developer to divide, in smaller units, complex software systems. At the same time the defined units should be functionally understandable in isolation, and finally it should be possible separate implementation level and architectural level concerns. The defined units can be successively reused in different "compositions" describing other systems.

- **Abstraction**
  "*It should be possible to describe the components and their interactions within software architecture in a way that clearly and explicitly prescribes the abstract roles in a system*". This property permits the description of a system in terms of roles of each elements in the structure but without introducing implementation details such as definition/use dependencies among the internal modules hidden behind the interfaces.

- **Re-usability**
  "*It should be possible to reuse components, connectors, and architectural patterns in different architectural descriptions, even if they were developed outside the context of the architectural system*". The reuse is one of the basic feature of component based software engineering, particularly important can be the reuse of architectural level elements with the possibility of further instantiation and refinement.

- **Heterogeneity**
  "*It should be possible to combine multiple, heterogeneous architectural descriptions*". Two different level of heterogeneity should be addressed. The first concerns the ability of combining different architectural patterns in a single system - e.g. components in a pipe-filter system can be conceived using a layered architecture. The second aspect of heterogeneity concerns the possible necessity of combining elements that will be implemented using different technologies. In fact as noted in [135], the choice of a specific technology, as for instance middleware, affects architectural descriptions. An ADL could usefully provide mechanisms to consider the integration of different technologies.

- **Configuration**
  "*Architectural descriptions should localize the description of system structure, independently of the elements being structured. They should also support dynamic reconfiguration*". As a consequence a language for architectural description should separate the definition of a composite structure from that of the constituting elements. Dynamic reconfiguration instead permits to describe the evolution of the architecture also at run-time.

- **Analysis**
  "*It should be possible to perform rich and varied analysis of architectural de-*

scriptions". This requirement address the ability to support automated and non automated reasoning about architectural description. Analysis can concern functional and non functional properties, such as performance and resource usage, and permit to reason about the effectiveness of the chosen architecture.

It has been observed that traditional programming languages present several weakness in the complete satisfaction of the properties listed above, lacking specific support for expressing architectural abstractions, forcing the researchers to start the development of specific languages for architectural description - generally referred as **ADL**. Major lack of traditional programming languages concern the low-level view that they generally provide of the interconnections among the components. These languages generally foresee, in fact, few communication paradigms for the definition of the intercommunication among the components that correspond to those directly defined in the language - as, for instance, procedure call or shared memory. Moreover with traditional languages the interconnections structure tend to disappear inside the definition of the computational elements (components), making harder the satisfaction of different properties such as, for instance, "composition". Finally traditional programming languages have a weak support for abstraction, generally requiring explicit definition of dependency among the different components (as the "include" statement of the C language), embedding in that way dependency relations inside the component definition. In [91] many other problems related to the use of traditional programming language for architectural description are discussed.

Most important feature of languages specifically thought for architectural description, has been the identification of two "first-class" elements such as **Components** and **Connectors**, promoting then the separate definition of the intercommunication protocols among the components. If a component is the element that contains computation and state of the system the connector is the element that specifies relations among the components and the medium that permits the interactions among the components. The identification of the connector as a first-class element is probably the main improvement and difference from traditional programming languages. Connectors description increases the expressiveness of intercommunication among components raising the complexity of the interactions that can be described. A sort of backbone can be produced in which the connector constitutes, as in hardware board, the intercommunication framework among the components that, in such manner, can be replaced with other functional equivalent components, maintaining the correct behavior of the system. An Architecture Description Language should provide primitive connectors and some mechanisms to create composite connectors, making it possible to define more and more complex interactions.

The provisioning of two different mechanisms for treating computation and communication is particularly useful considering that:

- connectors can be quite complex, and no single component is the right place in which to put this complexity;

- it must be possible to identify the definition of a connector as a whole. This task becomes particularly difficult if the definition of the communication paradigm is spread across several components;
- connectors define a template of interactions that could be instantiated more times in the same system;
- components can be used differently depending on the connector to which they are connected.

Several ADL have been defined and certainly further evolution will appear in the future. Below I report a short list of some important examples of ADL with references for the interested reader:

- Darwin [4, 115]
- Wright [21, 89]
- C2 [1, 119]
- Unicon [151]
- Rapide [14, 112]

All these languages provide suitable mechanisms for architectural description, even though they often satisfy the properties listed above only partially. Each one of them provides, however, some distinctive capabilities. For instance; C2 uses an event-based style to describe the interactions among the components; Darwin supports the analysis of distributed message passing system; Rapide provides tools for architecture simulation (to notice, it does not recognize connectors as first class elements); Unicon provides compilers for architectural design with heterogeneous component and connector types; Wright supports the formal specification and analysis of the interactions among architectural components and connectors. It uses CSP [100] as the basis for formally describing the behavior of components and the interactions among components described by connectors.

2.2.2. Architectural Styles

An important concept emerged in the field of SA is that of an *Architectural Style*. In particular, the question which architectural styles address, is how to leverage past experience to produce better design? The idea is to specify a precise idiom that characterizes a family of systems that are related by shared structural and semantic properties [20]. Many architectural styles have been used unconsciously by system developer over the years as system designers recognized the value of specific organizational principles and structures for certain classes of software. Certainly terminologies as *Client/Server architecture*, or *Pipeline architecture* and many other, have been defined as consequence of the necessity of communicating, in few words, complex concepts related to complex system architectures.

To really provide such conceptual leverage, architectural styles must define:

- A precise **vocabulary** of design elements (components and connectors) types.
- A set of **constraints** that define the permitted compositions of those elements. For example, the rules might prohibit cycles in a particular pipe-filter style, specify a n-to-one relationship in a client/server organization.
- **Semantic interpretation**, whereby compositions of design elements, suitably constrained be the design rules, have well-defined meanings
- **Analyses** that can be performed on systems built using that style. For instance an important kind of analysis that can be carried out given an architectural definition, can be deadlock detection and other behavioral properties for a client-server message passing system as illustrated by [102, 161].

In [91] a rich list of identified architectural styles with a short discussion of related advantages and disadvantages for each style is presented. They should constitute the basic instruments of a good software architect that should recognize the more appropriate architectural style for each system under development. Examples of architectural style are:

- Pipes and Filters architectures
- Blackboard architectures
- Publish-Subscribe or implicit invocation architecture
- Call and return architecture
- Layered architecture
- Client/Server architecture

2.3. The Unified Modeling Language (UML)

2.3.1. UML generic

UML [18] is an acronym that stands for Unified Modeling Language. Nowadays it is certainly the most famous and used language to define software models. It is a standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems. The UML represents a collection of best engineering practices that have proven successful in the modeling of large and complex systems. The UML is a very important part of developing object oriented software and the software development process. The UML uses mostly graphical notations to express design of software projects. Using the UML helps project teams communicate, explore potential designs, and validate the architectural design of the software.

The primary goals that drove the design of the UML were [55]:

1. Provide users with a ready-to-use, expressive visual modeling language so they can develop and exchange meaningful models.
2. Provide extensibility and specialization mechanisms to extend the core concepts.
3. Be independent of particular programming languages and development processes.
4. Provide a formal basis for understanding the modeling language.
5. Encourage the growth of the OO tools market.
6. Support higher-level development concepts such as collaborations, frameworks, patterns and components.
7. Integrate best practices.

In the following of this chapter I give a brief overview of the main features of the UML, considering that the upcoming specification of UML consists of four different documents for an amount of more than one thousand pages. In particular I focus the attention on the new mechanisms that have been added to the UML in the version 2.0. Until now UML have been generally used to model object-oriented system at a quite low level of abstraction. The new version instead will enclose mechanisms that have been mainly borrowed form the software architecture research field, transforming in that way UML in a powerful mechanism for modeling component-based software system. Another important requirement in the revision of UML has been the introduction of a precise semantics for each element, towards the real affirmation of the model driven development vision (that I illustrate in the last section of this chapter).

2.3.2. UML a Bit of History

Identifiable object-oriented modeling languages began to appear between mid-1970 and the late 1980s as various methodologists experimented with different approaches to object-oriented analysis and design. The number of identified modeling languages increased from less than 10 to more than 50 during the period between 1989-1994. Many users of OO methods had trouble finding complete satisfaction in any one modeling language, fueling the "method wars". By the mid-1990s, new iterations of these methods began to appear and these methods began to incorporate each others techniques, and a few clearly prominent methods emerged.

The development of UML began in late 1994 when Grady Booch and Jim Rumbaugh of Rational Software Corporation began their work on unifying the Booch and OMT (Object Modeling Technique) methods. In the Fall of 1995, Ivar Jacobson and his Objectory company joined Rational and this unification effort, merging in the OOSE (Object-Oriented Software Engineering) method.

The efforts of Booch, Rumbaugh, and Jacobson resulted in the release of the UML 0.9 and 0.91 documents in June and October of 1996. During 1996, the UML authors invited and received feedback from the general community. They incorporated this feedback, but it was clear that additional focused attention was still required.

While Rational was bringing UML together, efforts were being made on achieving the broader goal of an industry standard modeling language. In early 1995, Ivar Jacobson (then Chief Technology Officer of Objectory) and Richard Soley (then Chief Technology Officer of OMG) decided to push harder to achieve standardization in the methods marketplace. In June 1995, an OMG-hosted meeting of all major methodologists (or their representatives) resulted in the first worldwide agreement to seek methodology standards, under the aegis of the OMG process.

During 1996, it became clear that several organizations saw UML as strategic to their business. A Request for Proposal (RFP) issued by the Object Management Group (OMG) provided the catalyst for these organizations to join forces around producing a joint RFP response. Rational established the UML Partners consortium with several organizations willing to dedicate resources to work toward a strong UML 1.0 definition.

In January 1997 IBM, ObjecTime, Platinum Technology, Ptech, Taskon, Reich Technologies and Softeam also submitted separate RFP responses to the OMG. These companies joined the UML partners to contribute their ideas, and together the partners produced the revised UML 1.1 response. The focus of the UML 1.1 release was to improve the clarity of the UML 1.0 semantics and to incorporate contributions from the new partners. It was submitted to the OMG for their consideration and adopted in the fall of 1997.

Nowadays, after seven years, the UML is undergoing under a major revision that will finish with the forthcoming release of the UML 2.0 specification; currently under finalization stage.

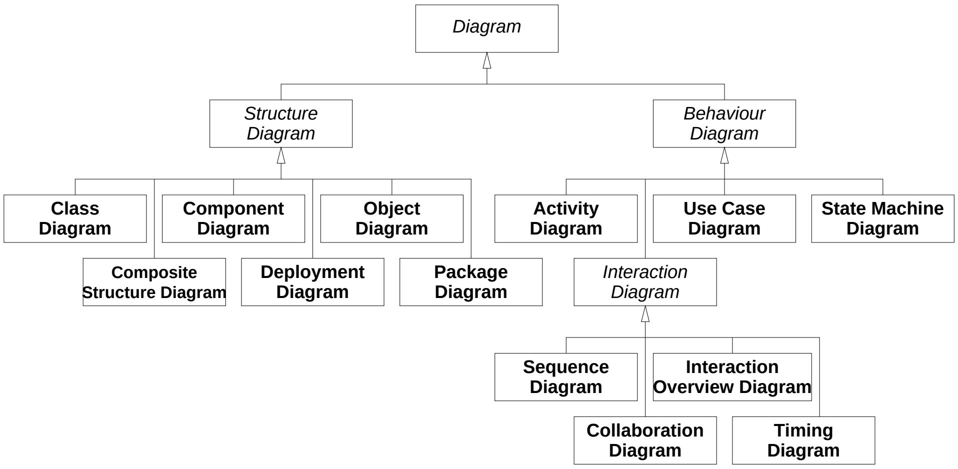

Figure 2.3.: UML Structure and Behavioral Diagram

2.3.3. UML Diagrams

Diagrams contain graphical elements arranged to illustrate a particular aspect of the system. A complete description of a system generally requires several diagrams of different type. Each diagram type can belong to one or more view depending on the contents of the diagram. The "4+1 model views" is not the only model that can be used to organize UML diagrams, even though this was the model that the UML's creators had in mind.

In the upcoming version 2, the UML provides thirteen different diagrams that can be used to specify a software model. Figure 2.3, from [169], provides a graphical view of how this diagrams can be organized in two different categories, such as:

- **Structure diagrams** mainly represent static concerns about, the system under development. The diagrams that are included in this class are: Class Diagram, Object Diagram, Component Diagram, Deployment Diagram, Package Diagram, and Composite Structure Diagram.

- **Behavioral diagrams** using the diagrams in this class the developer can specify behavioral concerns. In this category are included: Use Case Diagram, Activity Diagram, Statechart Diagram, and four different types of Interaction Diagram.

In the following we provide, for each diagram, a very short description, just to give a glimpse on the UML modeling power. The interested reader can refer to [81, 169] for more detailed information.

Class Diagram. A class diagram provides a static view on the relation among the classes in the system. Each class defines the properties and behavior of a type of object. Different kinds of relations can be defined between classes, such as: *associations*, *generalization*, *dependency*, and *abstraction* [81]. Graphically a class diagram is represented as a set of box, each box representing a class, and line connecting them, each line representing a relation between the class.

Object Diagram. An Object Diagram shows a snapshot of the system during its execution. It can be considered as a variant of the class diagram in which the relationship shown are only those actually active in a precise instant during execution.

Component Diagram. In the UML a component is an element from a logical architecture that encapsulates artifacts (source code or executable) that provide specific services. Therefore a component diagram shows in a single picture more components and their respective relations.

Deployment Diagram. The deployment diagram depicts the runtime architecture of devices, execution environments, and artifacts that reside in this architecture. It is the ultimate physical description of the system topology, describing the structure of the hardware units and the software that executes on each unit. Graphically a deployment diagram contains a set of boxes, representing computational resources upon which artifacts may be deployed and the respective deployed artifacts, and line connecting the node, representing communication paths.

Package Diagram. A package provides a grouping mechanism for organizing UML elements. In UML, a package is used to group elements and to provide a namespace for the grouped elements. Graphically a package is represented as the classical folder symbol, with the name of the package specified in the upper-left smaller rectangle. Inside the "folder" the specific content is represented. It is possible to have a hierarchy of packages. In a package diagram one or more packages are represented with corresponding relations within them.

Composite Structure Diagram. The composite structure diagram provides a mechanism to describe the connections between the elements that work together within a particular classifier such as a use case, object, collaboration, class, or activity.

Use Case Diagram. Use-case diagrams shows the relations among the external actors and the functionality provided by the system. The description of the use case is generally done using a textual form. Different from have been proposed for the textual description each one foreseeing different class of entry. Probably the most used proposal is that of Cockburn [65]. A use case diagram is graphically represented as a set of stick man, representing the actors, a set of oval, representing the use cases, and line connecting the use cases to the actors interested in the specific uses case. Relation, such as defined for the classes, can be defined among actors and use cases.

Statechart Diagram. A state machine can be a complement to the description of a class. It shows all the possible states that can be assumed by the object of the class during it life cycle. At the same time the state machine report the event that has triggered the state transition. The UML 2 define two different kind of state machine:

- the ***behavioral state machine*** describes all the details of a class life cycle.
- the ***protocol state machine*** focuses only on the transition of states and rules governing the execution order of operations. This kind of state machine can provide guidelines for the implementation of a specific interface. At the same time the protocol state machine define the rules that must be followed by external elements to correctly interact with the described class (component). Clearly the introduction of this kind of state machine fosters the adoption of the UML for the description of component based software.

Graphically a state machine is represented as a set of node (the states) and directed edge (transitions). Each edge is generally marked with the event that has caused the state transition.

Activity Diagram. An Activity Diagram shows a sequential flow of actions. It is used to describe the activities performed in the a general process workflow. Typical situation in which an activity diagram is used, are:

- to capture the activities that will be performed when an operation is executing.
- To capture the internal actions carried out in an object when a methods is invoked
- To show how a set of related actions can be performed and how they affect objects around them
- To show how an instance of a use case can be performed in terms of actions and object state changes
- To show how a business works in term of workers, workflows, organization, and objects.

Sequence Diagram. A sequence diagram is a particular kind of interaction diagram, it shows a dynamic collaboration between a number of objects, in terms of the messages that this two objects have exchanged during a period of time. Graphically a sequence diagram is represented with a line of rectangles, representing the objects involved, and with vertical line attached to each rectangle, representing the life-line of the associated object. Within the life-line arrows represent messages exchanged among the objects. Starting from the last version of the UML it is possible to give further semantics to the diagram representing possible behaviors such as alternative messages, parallel messages, loops, critical regions, weak sequencing, strict sequencing and other.

Communication Diagram. The communication diagram is another kind of interaction diagram, that results to be similar to a sequence diagram. However communication diagram focus the attention also on the existing relations among the collaborating objects. The sequencing of messages is given through a sequence numbering scheme. Graphically a communication diagram contains rectangle, representing the collaborating objects, that are linked by line, representing the existence of a relations between the objects. Finally a small arrow is placed alongside the line for each message that is being modeled between the two objects, the arrow is labeled with a number that express the order following the numbering scheme.

Interaction Overview Diagram. An interaction overview diagram define interactions through a variant of activity diagrams where the nodes are actually interactions.

Timing Diagram. Timing diagrams are used to show interactions among objects when the primary objective is to reason about time. Timing diagrams focus on conditions changing within an among life-lines along a linear time axis.

It is worth to note that it is possible to extend the set of elements that can be used within a diagram using the extension mechanisms that the language provide. The language provides three standard way to extend the set of elements available, tagged values, constraints and stereotype. Using this mechanisms the modeler can use the predefined elements to create new elements with added semantics. An important tool for defining extension is the Object Constraint Language (OCL) [172], that is a language defined by the OMG that permit to express logical constraint on the element of a diagrams. Another alternative to extend the UML is to intervene at the meta level defining in such manner a UML like notation.

2.4. Using UML for Software Architecture and Component-based System Modeling

The UML has been initially defined to aid the development of object-oriented systems, providing visual representation for OO related concepts. From the beginning it was clear that the language could be used to describe any kind of systems, even non software systems. Thanks to the extension mechanisms that the UML provides, several initiatives started trying to adapt/extend UML for use in different development fields. The increasing availability of tools for diagrams management and the spreading diffusion of the language, pushed other modeling communities, such as the SA community, to study the possible use of UML for architecture description, even though the first versions of the UML did not provide a precise and clear semantic for the elements in the diagrams. An obvious strong relation exists between the UML and the software architecture, being both two instruments for software modeling. However the use of the UML for software architecture description, even though it can appear quite natural, is not so obvious. Three main concepts, whose importance has been assessed by the software architecture community, are not directly supported by the UML. In particular the UML lacks direct supports for modeling and exploiting architectural styles, explicit software connectors, and local and global constraints. Even though the upcoming version 2 of the UML provide a better support for describing connectors, such as ports and protocol state machines, there is not yet a complete inclusion of the software architecture abstractions, such as those discussed in Section 2.2, in the UML. Then two different approaches can be thought for representing software architecture using the UML:

- use the extension mechanisms to define constraints on the UML meta model
- extend the UML meta model defining new elements to represent the lacking abstractions

Of this two options only the first can be considered practical. The second option, in fact, disable all the advantages of using the UML for architectural description. The resulting language , in fact, will not be anymore standard and all the extension will not be understood by any one of the UML compliant tool. Certainly the most extensive work on this topic is that of Medvidovic et al. [120]. In their work the authors proposed possible extensions to the UML. In particular they successfully extended the UML to represent software architectures following the C2 [1], Wright [21], and Rapide [14] styles.

2.4.1. UML Components

"UML Component" is a proposal for using the UML to model component based software systems. It has been one of the the first proposal focused on the use of the UML for component-based system description, and it has been described by Cheesman and Daniels in [63]. The idea, underneath this proposal, is to increase the number of elements that can be suitably used to represent software components via the definition of particular stereotypes. However the "UML Component" proposal seems to me having big lack for what concerns the support for architectural concepts such those highlighted by the Software Architecture community (e.g. connectors), and certainly is nowadays overtaken by the advent of the UML version 2, that as illustrated in the previous section, provide better mechanisms for CBS description. However in their book Cheesman and Daniels illustrate an interesting iterative process, illustrated in a fairly intuitive way in Figure 2.4, for the development of the component based software system that can produce good results.

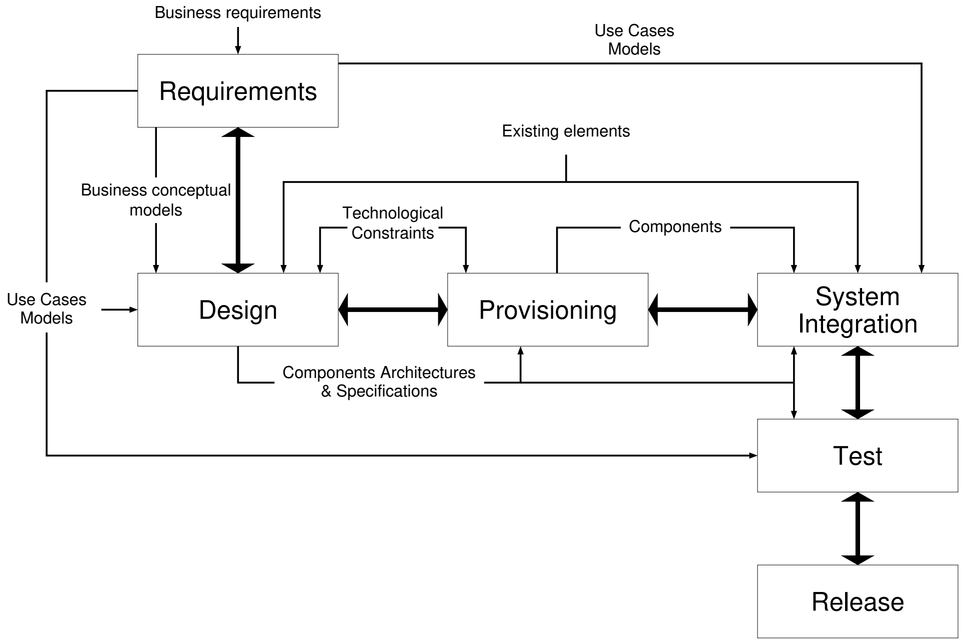

Figure 2.4.: "UML Component" Development Process (from [63])

2.5. Model Driven Architecture (MDA)

The most advanced frontier in the use of models in software systems production is certainly that prefigured by the Model Driven Development (MDD) methods. The

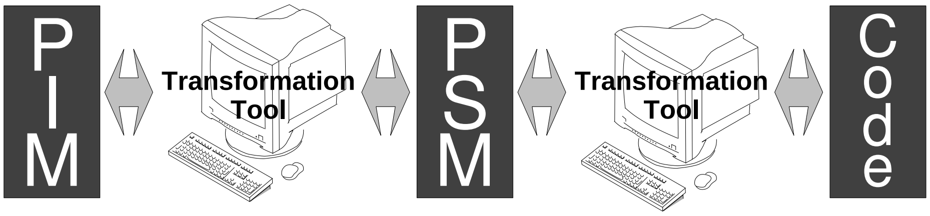

Figure 2.5.: The three main steps in the MDA process

idea underneath this visionary approach is the possibility of having models automatically transformed into the corresponding final executable code. As a result a new generation of languages and tools will be available to the system developer, providing a level of abstraction never experimented until now. The benefits of such an approach are obviously numerous, most important, from an architectural point of view is the possibility of automatically verifying models on a computer, that will be successively better refined to become the final program. In Chapter 9 I present an approach, developed in cooperation with Giovanni Denaro and Wolfgang Emmerich, that will certainly benefit from the availability of tools suitable for the MDD analysis, in particular for what concerns the performance analysis. The importance of having tools for the analysis of the system architecture from the first phases of the development has been generally recognized as one of the basic factors to reduce the risk of missing system requirements in the final system implementation, with corresponding loose of big quantity of money, since the fundamental decisions are taken in the early stage of the development.

The most concrete effort towards the realization of the MDD view is certainly the Model Driven Architecture (MDA) initiative. As defined by the OMG, that is the no profit organization that has launched and is driving the project, MDA is a way to organize and manage enterprise architectures supported by automated tools and services for both defining the models and facilitating transformations between different models types. MDA is based on the use of the forthcoming version of the Unified Modeling Language that will be based on a precise semantic that will enable the unambiguous transformation of software models. A basic transformation in MDA is that between the Platform Independent Model (PIM) and the Platform Specific Model (PSM).

the Platform Independent Model. The PIM is the first model that will be defined during the modeling of a system following the MDA method. A the name says it does not contain any particular reference to a specific platform on which the system will be successively deployed. Developing the model the architect will concentrate his/her attention to how the system can better support the particular business for which it

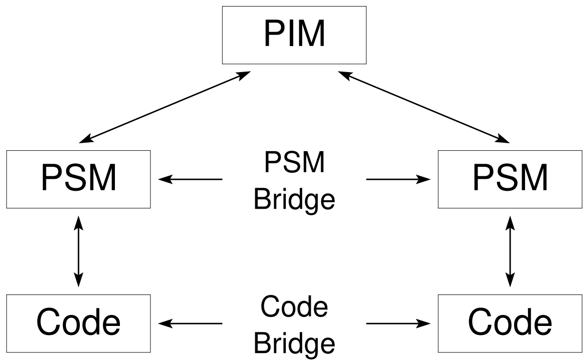

Figure 2.6.: MDA interoperability using bridges [104]

is developed.

the Platform Specific Model. When the PIM has been defined the next step will be to transform it in the PSM. This model adds to the PIM all the specific information that will better adapt the model to a specific platform. For instance an Enterprise Java Beans (EJB - see Chapter 3) PSM is a model of the system in terms of EJB structures. From the PSM it will be possible to directly and automatically obtain the final code/implementation.

In the MDA view the transformation between the different models will be executed in automatic with the support of specific tools following the steps represented in Figure 2.5. Since the level of detail of the PSM is fairly high the automatic transformation into the final code is not a so difficult and new task. The real challenge of the MDA approach is the transformation between the PIM and PSM. There is not a wide number of tools enabling the MDA view, yet, however considering the benefits that the real success of this methodology to software production will bring it is not difficult to foresee that many tools will be released in the near future (first examples of MDA tool are emerging, in [11] a short list is provided).

Among the several benefits that the MDA development process will bring such as **productivity**, **maintenance**, and **documentation**, of particular importance are certainly **portability** and **interoperability** that have nowadays great importance since the continue integration of different systems. In fact as Figure 2.6 well illustrate the PIM model can be transformed in different PSM and at the same time this models contains all the information that can be automatically used by suitable tools to generate bridges permitting the easier interoperability of different parts of code deployed on different platforms.

3. Implementing Component-based Software Systems

The previous chapter gave an overview of the most advanced methodologies and languages developed to describe complex software systems. However, the development of modern software systems probably would not even possible without the great development in the area of software technologies that we had in the last years. We can certainly affirm, in fact, that the area of software tools and technologies to aid the development of complex software systems is far more mature then the correspondingly modeling tools and languages.

Among the technologies defined to assist the implementation of complex software system, two of them received a lot of interest from research and industry communities. They are:

- **middleware**
- **component based software models**

These technologies are strongly interrelated and nowadays commercial products, such as J2EE and .NET [150, 12], provide mechanisms of both categories. As a general rule we can distinguish middleware functionalities as those addressing interoperability and distributions issues, instead component models mainly focus on reuse issues, and define rules for packaging and for accessing to services. In the following I separately discuss middleware and component-based software highlighting which have been the motivations that lead to their development and for each area some of the different models and solutions proposed.

3.1. Middleware

In the last years many different causes contributed to the increasing demand for distributed software systems. Particularly relevant are both the recent trend to have company spread in many different places, also as consequence of the great number of companies merging, and the advent of the World Wide Web that led to the creation of "e-facilities", i.e. to the possibility of creating a set of electronic services that is going to have a great impact on the way in which services (provided by government or commercial companies) are going to be provided and used. In the development of a new application some general non-functional requirements/features lead to the choice of a distributed architecture, instead of a centralized one, in particular the most important are [76]:

1. **Scalability**: The system must be capable of accommodating growing load in the future. Distributed system can be "easily" scalable adding new computers in the original configuration;

2. **Resource Sharing**: Often is necessary to share hardware, software and data;

3. **Heterogeneity**: Use of different technology for the implementation of services and availability of legacy components (differences can be present at the level of programming languages, operating systems, hardware platforms, network protocols);

4. **Fault-Tolerance**: Operations should continue also in presence of faults. Generally obtained by means of redundant components;

5. **Openness**: Distributed system can be easily extended and modified with new functionalities.

It is worth noting that performance issues it is not per se a motivation that should lead the software engineer to the choice of a distributed architecture. In fact, even though the possibility of having real parallelism through the use of more real processors, generally not available on single PCs, the benefits obtainable could be spoiled by the higher cost (up to 2000 times more) of interprocess communications. In his book [76] Emmerich observes that the option for a distributed system should be a “last chance option”, in the sense that it is better to avoid it if not really necessary. In fact development difficulties and therefore costs are considerably higher in the development of a distributed system.

In [152] the authors provide the following definition for a distributed system:

> *A distributed system is a collection of processors that do not share memory or a clock. Instead, each processor has its own local memory, and the processors communicate with each other through various communication lines. The processors in a distributed system vary in size and function. They may include small microprocessor, workstations, minicomputers, and large general purpose computer systems.*
>
> ...
>
> *A distributed system must provide various mechanisms for process synchronization and communication, for dealing with the deadlock problem, and for dealing with a variety of failures that are not encountered in a centralized system.*

The easy possibility of connecting different hardware systems, the trend in companies development and the emerging requirements, listed above, enormously raised, in the last years, the need for distributed software systems. However the development of such complex software systems requires the availability of tools and technologies that make it a "feasible the task". The development of a distributed software is, in fact, far more complex than the development of a centralized one. For this reason it is useful to hide, as much as possible, to system developers the issues strictly related to distribution providing higher level services, and permitting to them to concentrate the attention on the business logic of the application. This is the rational for the development of what is generally referred as **middleware**. As a general idea middleware is a kind of connectivity software that allows applications and users to interact with each other across a network. However it is hard to define middleware in a technically precise way. Middleware components have several properties that, taken together, usually make clear that a component is not an application or platform specific service. In particular middleware component should provide services that are generic across applications and industries, that run on multiple platforms, that are distributed, and that support standard interfaces and protocols [33]. For the sake of clarity can be useful to further extend the discussion and provide some example of middleware starting from the requirements above.

A middleware service meets the need of a wide variety of applications across many industries. For example a message switch that translate messages between different formats, is considered middleware if it makes it easy to add new formats and is usable by many applications. If it deals with formats only for a single industry and is embedded in a single application, then it is not middleware. The second properties requires that middleware service must have implementation that run on multiple platforms. Otherwise, it is a platform service. An example of middleware are Data Base Management Systems (DBMSs). To have good platform coverage, middleware services are usually programmed to be portable with minimal and predictable effort. The third properties requires that a middleware service must be distributed. That is, it either can be accessed remotely or enables other services and applications to be accessed remotely. A remotely accessible middleware usually includes a client part, which supports the service's Application Programming Interface (API) running in the application's address space, and a server part, which support the service's main functions and may run in a different address space. Finally the fourth properties requires that a middleware service supports a standard protocol or at least a published one. That way, multiple implementations of the service can be developed and those implementations will cooperate. Moreover a middleware service should support a standard API. A middleware service is transparent with respect to an API if it can be accessed via that API without modifying that API.

Figure 3.1, from [77], wants to give a graphical representation of the description provided above, suggesting that middleware can be considered in respect to the ISO/OSI

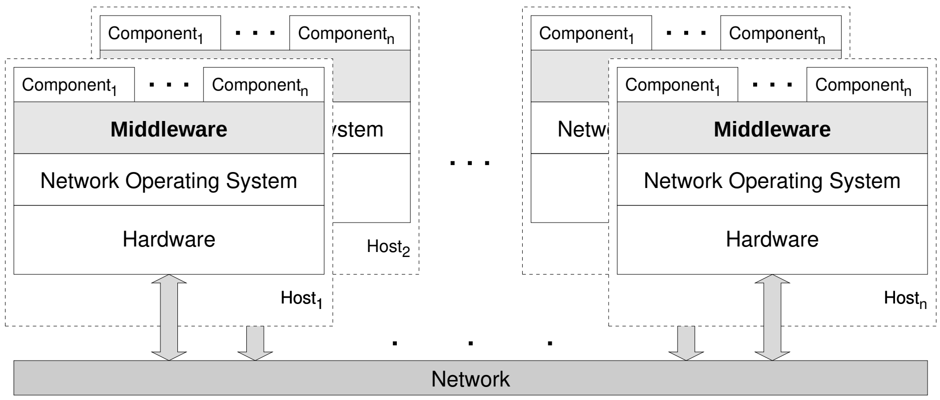

Figure 3.1.: Middleware in Distributed System Construction

standard as mainly providing 5th and 6th layer services. Figure 3.2, from [165], gives example of services that are generally considered or not middleware (in the figure the hierarchy among the middleware layers do not imply a real relation).

Main objective of middleware is to make transparent, to the average application engineer, the new dimensions of complexity introduced by distributed systems. Emmerich [76] identifies eight main dimensions for desirable transparency features. The considered dimensions are not necessarily orthogonal and could not be possible to pursue one of them without considering some other. The eight dimensions listed in [76], that corresponds to new issues introduced by the distribution, are:

1. **Access Transparency**, the interface to a service do not depends from the location of the components that use it. Otherwise is not an easy task to move the service to a different host;

2. **Location Transparency**, a request for a service can be made without knowing the physical location of the components that provide the service. Otherwise moving components becomes almost impossible;

3. **Migration Transparency**, components can be migrated to different host without that the user are aware of that, and that the developer of clients components take a special consideration;

4. **Replication Transparency**, the user of a service and the application programmer are not aware that a service they are using are provided by a replica;

5. **Concurrency Transparency**, several components may concurrently request services from a shared component while the shared component's integrity is pre-

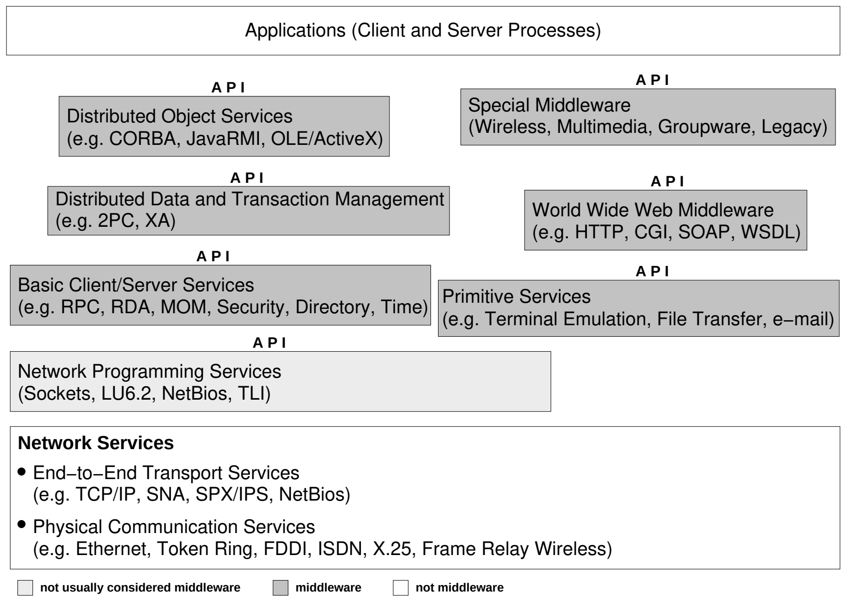

Figure 3.2.: Middleware examples (among middleware elements, hierarchy is not meaningful)

served and neither users nor application engineers have to see how concurrency is controlled;

6. **Scalability Transparency**, to the users and designers is transparent how the system scales to accommodate a growing load;

7. **Performance Transparency**, the users and the application programmers are not aware of how the system performance is actually achieved;

8. **Failure Transparency**, the user and the application programmer are unaware of how the system hides the failure. They believe that the service cannot fail;

Figure 3.3 shows a kind of "X is necessary for Y" relation among these different kinds of transparency

3.1.1. Distributed Computing Models

The restriction of no shared memory and information exchange through messages is of key importance in distributed computing system and it makes the difference between this kind of systems and shared memory multiprocessor computing systems.

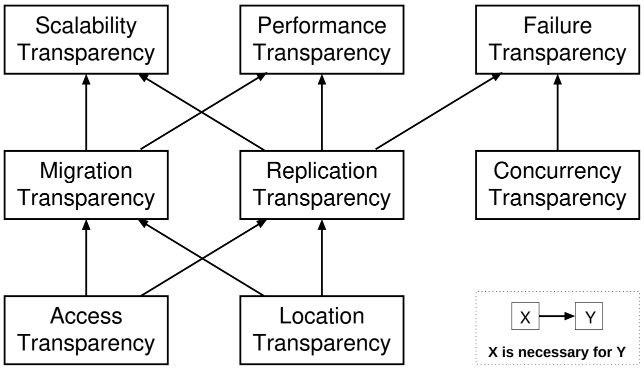

Figure 3.3.: Dimensions of Transparency and Relations among Dimensions (from [76])

Distributed computing can be logically achieved through one or more of the following [165] model:

1. **File Transfer Model**: this was one of the first model trying to exploit the resources distribution. Following this paradigm the applications, that need data residing in another machine, make the log-on on it and then transfer the data. Therefore, when the transferring is ended, the foreseen computations are performed off-line. In this model the distribution regards only the data and it is applicable only with low load and low concurrency.

2. **Client/Server Model**: Client/server model is a concept for describing communications between computing processes that are classified as service consumers (clients) and service providers (servers). In this context C/S refers to a software architecture and not to an hardware architecture.

3. **Peer-to-Peer Model**: following this paradigm more processes, located on different machines, cooperate to reach the solution of the problem. In different moments a process perform both server and client duties. This paradigm is receiving more and more interest from the scientific community, also because it permits to reach great computation power collecting the spare CPU cycles of voluntary users sitting in front of their desktop. Interesting campaigns have been promoted using this distributed paradigm such as the "Intel Philanthropic Peer-to-Peer Program" [8].

The Client/Server model is certainly the most used paradigm for business applications, nowadays, and for this reason I provide in the following some more information on it, instead the discussion on the other two paradigms it will not be further extended.

Client/Server Computing. Client/Server computing attempts to leverage the capabilities of the network used by a typical corporations, that are composed of many relatively powerful workstations and a limited number of dedicated servers. Its importance is growing up enormously in the last years, also thanks to the advent of the World Wide Web and related technologies, such as web servers, browsers, HTTP and so on. In a client/server application it is possible to distinguish at least among three different levels of computation:

- *Graphical interface*: it concerns the final user interface and how the results of the computations will be provided to the user;
- *Business logic*: it can be considered the core of the specific application and concerns the logic for what the application has been developed for;
- *Data Management*: it concerns the retrieval of the data relevant for the application.

The different levels of computation listed above can be differently structured on different tiers and depending on this distribution different client/server architectures have been identified. In particular we can have [107]:

- **two-tier architectures** in which the three logical levels are distributed among the client and the server with a further distinction among fat-client/thin-server architecture, in which the business logic is mainly put in the client, and thin-client/fat-server, in which the business logic is mainly put in the server.
- **three-tier architectures** in which each logical level is reflected in the application architecture that contain a different tier for each level. Standard Web applications are probably the most common example of three-tier systems.
- **n-tier architectures**, that can be considered an extension of the previous one in which the business logic is further split in more tiers.

The organization on more tiers of an application obviously open the opportunity for an easier reuse of part of the application itself, and is the reference architecture for web based applications developed using modern framework such as .Net [12] and J2EE [6].

3.1.2. Types of Middleware

Above I discussed the motivations that lead to the choice of a distributed application and the model that can be used to logically organize a distributed applications. In this section, instead, I briefly introduce the technologies that have been proposed to enable and make easier the development of distributed software (i.e. middleware technologies). In particular middleware technologies, for implementing Client/Server applications, are generally divided in four main categories, such as transactional, message oriented, procedural, object based middleware. An interesting comparison and discussion of the points of strength and weakness of the different technologies can be found in [77].

Transactional Middleware. A transaction is a unit of work that execute exactly once and produces permanent results. Transaction oriented middleware provide mechanisms that make easier the implementation of applications that requires the execution of transaction on distributed systems/databases. Often transaction is referred to have "ACID" properties, that means that a transaction should have the following properties:

- **Atomicity**: A transaction allows for the grouping of one or more changes to tables and rows in the database to form an atomic or indivisible operation. That is, either all of the changes occur or none of them do. If for any reason the transaction cannot be completed, everything this transaction changed can be restored to the state it was in prior to the start of the transaction via a rollback operation.
- **Consistency**: Transactions always operate on a consistent view of the data and when they end always leave the data in a consistent state. Data may be said to be consistent as long as it conforms to a set of invariants, such as no two rows in the customer table have the same customer id and all orders have an associated customer row. While a transaction executes these invariants may be violated, but no other transaction will be allowed to see these inconsistencies, and all such inconsistencies will have been eliminated by the time the transaction ends.
- **Isolation**: To a given transaction, it should appear as though it is running all by itself on the database. The effects of concurrently running transactions are invisible to this transaction, and the effects of this transaction are invisible to others until the transaction is committed.
- **Durability**: Once a transaction is committed, its effects are guaranteed to persist even in the event of subsequent system failures. Until the transaction commits, not only are any changes made by that transaction not durable, but are guaranteed not to persist in the face of a system failure, as crash recovery will rollback their effects.

An important example of this kind of middleware are Transaction Processing (TP) Monitors, that are deeply described in [32]. TP monitor is a technology emerged more than 30 years ago that provides functionalities that easily permit to the system developer to implement an application with support for distributed transactions. The

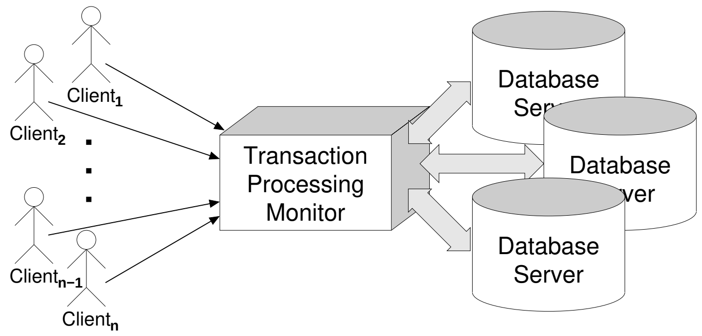

Figure 3.4.: Transaction Processing Monitor

basic idea is to introduce a layer among databases and clients that works as a monitor for transactions execution. The it accepts messages specifying transactions and queues them for processing them against the database. Figure 3.4 gives a graphical idea of how transaction monitor systems are structured. A implementation of TP monitor should accomplish to several tasks, in particular a TP monitor should manage processes, should provide an interprocess communication abstraction that hides networking details, and should help system managers efficiently an easily control large networks of processor and terminals. Then the main function of a TP monitor is to coordinate the flow of transaction requests between terminals or other devices and application programs that can process these requests.

Message Oriented Middleware (MOM). Message Oriented Middleware supports the communications between processes in a distributed application providing mechanisms for exchanging messages. Generally MOM middleware implements queue between interoperating processes, so if the destination is busy, the message is held in a temporary storage until it can be processed. This means that with MOM clients and servers can proceed concurrently without the necessity for the client to block waiting for server response. MOMs generally support multi-cast messaging permitting the distribution of the same message to more than one receiver. With the advent of mobile devices the importance of this kind of middleware grow up since it is well suited for implementing distributed event notification [72], and publish/subscribe-based architecture [61].

Procedural Middleware. The idea of Remote Procedure Calls (RPC) is quite simple. It is based on the observation that procedure call is a well-known and well-understood mechanism for transferring control and data within a program running on a computer. Therefore, it is proposed that the same mechanism could be extended to provide for

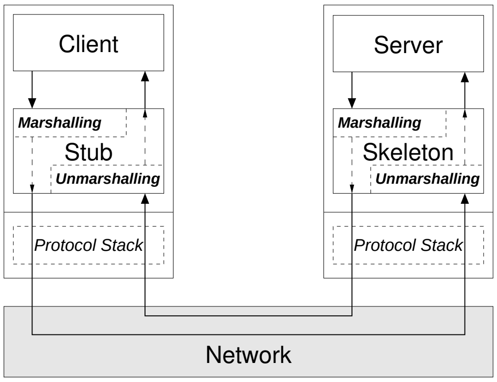

Figure 3.5.: Typical components in a Remote Procedure Call Implementation

transferring of control and data across a communication network [50]. The idea behind the implementation of a RPC mechanism is outlined in Figure 3.5. In the figure a fake version of the service is available on the client side (the **stub**). When the service is invoked the stub component packs the data and sends it to the corresponding component on the remote machine (the **skeleton**). The skeleton unpacks the data, invokes the actual service and returns the results to the stub (and from this element the data will be received by the original client). The generation of the couple stub/skeleton and the underlining communication mechanisms is hidden to the application developer.

Object-based Middleware. Object middleware can be considered as an evolution of RPC middleware. The intention in this case it to bring into the distribution environment the object method invocation paradigm, permitting to an object in one machine to invoke a method of another object stored in the memory of another machine. Object middleware manages a remote method invocation in a similar way of what illustrated in Figure 3.5 for RPC. In object middleware, the distributed software that provide services for remote method invocation is referred as Object Request Broker (**ORB**). The relevant functions of an ORB technology are:

- the **Interface Definition Language (IDE)**: this is a language that permit to define the service provided by a remote object and to generate local stub and skeleton components. In particular the language and the associated compilers permit to solve language heterogeneity enabling the cooperation among objects defined using different languages.
- location and possible activation of remote objects: this refer to services provided

by ORB to retrieve remote object location and to activate remote object on the remote machine.

- communication between clients and objects: ORB enables, using the services and components described in the previous two points, and the mechanisms from the network layers below, the communications among remote objects.

ORB technology is nowadays a quite mature discipline and three main products have been developed by consortia/industries. In particular the first notable proposal has been the Common Object Request Broker Architecture (**CORBA** [15]) that is a specification released by the Object Management Group and that has had a great success with many companies that have provided their own implementation [5]. Sun Microsystems proposed an ORB for Java based application (no language heterogeneity is supported in this case) that have been called Java Remote Method Invocation (**Java RMI**) [9]. Finally Microsoft has developed the Distributed COM architecture (**DCOM**), that permits to an object inside a COM component (COM is the component model developer by Microsoft - see Section 3.2 for further details) to invoke services from another distributed component [2]. In [76] an accurate discussion on the subject of object middleware is provided.

3.1.3. A General Note on Transparency

So far we discussed technologies that permit to simplify the development of distributed applications providing high level services that free the developer from knowing too much about distribution details. However, it is worth noting that it is not in general possible, and sometimes neither desirable, to completely hide the nature of distribution to the developer, since this could lead to a systems wrongly structured and dimensioned, and to scarce performance. For instance for the particular case of OO middleware many different motivations suggest the importance of making informed choices concerning distribution issues. In particular we can enclose in the list the following elements [170, 76]:

1. **Life Cycle**: the *creation* of a remote object cannot be performed using a standard constructor that create an object in the same address space of the invoking object. This requires the implementation of some service such as a factory service that will create and return an object reference for a specified type on a remote machine. Another problem that is not present with local objects is *migration*. Migration refers to the possibility that at some point in time an object could move to another host. In this case heterogeneity problem will rise. Finally differences between the two paradigm are evident for the *deletion* of the objects. In particular it is in general difficult to implement distributed garbage collections algorithms. Referential integrity is another rather expensive task.

It is, in fact, difficult to know all the existing references to an object, since it can be passed around among the objects in the system, without a central point that can trace the added reference. With distributed objects could be necessary to deal with server objects no more available because for instance the server in which the object was residing, went down. Some of the mentioned problems could require the direct involvement of the application developer to manage the different situations.

2. **Activation**: in distributed OO programming new issues such as:

   a) machines hosting server objects could be sometimes stopped and restarted
   b) resources required by all the server objects on a host may be greater than the resources available
   c) server object could be idle for long time between two invocations than could be opportune do not waste resources

   requires the introduction of mechanisms to “hibernate” object state and to resume them when a call for the object arrives. Middleware generally hide this behavior from the developer but sometimes, also in order to save memory space, the engineer could be interested in managing activation and deactivation of remote objects explicitly.

3. **Latency**: in [76] the author reports that a local call requires, in modern workstations, 250 nanoseconds instead a remote request could require between 0.1 and 10 milliseconds. Therefore a remote request is about 2,000 times more expensive than a local one. This fact suggests that the location of an object could become a design problem that requires a careful evaluation by the system developer. In particular could be important to locate objects that communicate a lot between them on the same machine.

4. **Memory Access**: remote references are quite big data structures, in complex middleware the necessity of space for a reference can be 100 times bigger than in "normal" OO programming. As consequence an application could not be able to maintain a lot of reference to remote objects. Engineer should consider memory issues caused by distribution and in some cases they should try to reduce the number of objects in the systems.

5. **Concurrency**: access to server objects must be controlled since real parallelism become available independently from the use of threads.

6. **Failures**: distributed objects have to deal with major probability of failures. In particular new kinds of failures can occur. A special treatment is required by the occurrence of **Partial Failures**. These are special kind o failures in

which an operation is executed by the server object but as consequence of a failure, for instance in the network device, the confirmation of the execution is not returned to the invoking object that then could think that the operation has not been performed. Middleware generally permit to associate different kinds of semantic to a distributed request, such as exactly once, at most once, maybe,at least once and atomic. Obviously more is assumed on middleware reliability more the service will cost in terms of resources and time (differently local call assume always an exactly once semantic).

7. **Communications**: due to the many cause of delay that can affect an invocation, could be, in some cases, more appropriate to use non-blocking calls. At the same time could be useful, for saving time, to have and use multicast invocations. This kind of communications, that are not generally foreseen by local paradigm, permit to collect in a single invocation more remote invocations and then to receive the results as soon as they are made available.

8. **Security**: distributed objects use the network for communications. Centralized applications trust that the user will not make the session available to unauthorized users, instead in distributed applications each request might require authenticated sessions.

3.2. Component Based Software Models

3.2.1. CBSE Basic Concepts

The nature of component should be intuitively clear from the name, that suggest that components indicates some kind of entities that should be used to be composed. Fortunately the intuitive idea reflects the reality and component are actually pieces of code, which complexity can be varying, that should be "easily" assembled with other components originating a more complex system. It is interesting to note that anything that can be done with components can be done without component, the difference is in the time that we need to reach the final implementation [160]. The last observation clarify the strong attention that the software community reserved to this topic in the last years and the current pressure toward the "componetization" of software production. Even though the concepts expressed above could seem fairly clear and intuitive, a wide agreement on the concepts leading to components and on a general definition of components still has to emerge. One of the most accepted and reported definition can be found in [160]:

> *A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component*

can be deployed independently and is subject to composition by third parties.

This definition covers different peculiar aspects of components. In particular it has a technical part, with aspects such as independence, contractual interfaces, and composition. It also has a market related part, with aspect such as third parties and deployment. It is worth to highlight the differences between the concept of component and that of object. The first think that should be noted is that a component is not a running elements, as an object, but rather a static elements that can be deployed on a system. Moreover a component should not depend from other specific components and could be deployed independently from them. Objects, instead, generally require services from other specific objects and this relation is embedded within the object itself. However, given this and other differences, it is certainly true that components and objects are strongly interrelated and they share many characteristics, moreover components are generally developed using object oriented concepts and languages and then at run time they take the form of a collection of objects.

With reference to the work that is presented in the next chapters, components can be classified considering as critical characteristic the capacity for the external user (the third part in the definition) of looking inside the component. Authors generally use the terms of black- gray- and white-box components with reference to different level of closure of the component internal essence. In particular a black-box component do not discover anything about its internal implementation. At the opposite end side of this spectrum, instead, a white-box component completely shows to the user its internal implementation. In the middle we can have different levels of gray-box components depending on how much details are made public to the user. The discussion about the opportunity of using one or the other of these different kinds of transparency is endless. However it is important noting that, even though the knowledge of implementation details could be useful, for example for analysis purpose, it should not lead per se to the selection of one component in place of another. The risk of making choices that will bind the system to a specific implementation of a component, loosing the possibility of successive easy substitutions of the component with other implementation, is, in fact, fairly high. Even though the classification that we discussed here take as discriminating factor the possibility for the user of accessing to implementation details, another dimension of this concept can be conceived considering software components. Experiences suggest, in fact, that it is a good practice to attach to a software component either information revealing specific component properties, or simplified models of the component itself, or any other detail that could improve the understanding on component behavior to the component user (i.e. system assembler). In this sense the classification above can be extended considering the quantity of information made available to the system assembler.

The **Interface** is another important concept cited in the definition above. An interface of a component defines the access point to a service provided by the component itself. A single component could provide different services specifying different interfaces. It is interesting noting that the choice of the interface to implement is of basic importance, also from a pure market point of view. The service that a component provides should be necessary to someone else otherwise the component has no market and there is no reason in developing it. In this context also aspects of granularity play an important role. The component should provide services sufficiently complex to justify the existence of a component. Simple component, in fact, can be more easily developed in house by system assembler. However to much complex services reduce the market for the component and therefore the convenience in developing it. The book of Szyperski et al. [160] is probably the best book on components with reference to the discussion of market aspects of component based engineering. The interested reader can certainly found in it many other interesting arguments related to the commercial nature of components.

In an ideal component world an interface should be completely characterized by a description that provides to the system assembler, a precise and complete information on the service that is implemented by the component. For instance using some formal mechanism. However, this is not the case, since the specification of a complex service is not easy to produce using such formalisms. At the same time could not be easy to check if the specification of a component matches with that for the component searched by the system assembler. An interesting and quite successful way to associate semantic to an interface is the use of **contracts**. The use of contracts in the development of object oriented system has been strongly promoted by Meyer [125], and also an object oriented language, such as Eiffel [124], has been developed with specific supports for defining contracts. A contract describes a services using first order logic and specifying condition that should hold before the invocation of the services and condition that will be true after the execution of the service. At the same time a contract can specify invariant condition that remain true during the whole execution of the service. Contracts seem to be a really useful mechanism in component based development and many research focus on the use of it for many different objectives. However the use of contract can raise some problems in particular when callbacks are considered. Different mechanisms and disciplines have been proposed to address this problem (this topic is extensively discussed in [160]), that however highlights the extreme difficulty in having a simple formalism for providing guarantees, to the system assembler, on component behavior when it is deployed in the final system.

3.2.2. Component Models and Platforms

In this section I briefly give some information on component models, that refer to all the specific technologies that provide mechanisms for easily bind together software components. Basically all these technologies provide a set of services that permit to produce and use components responding to the definition of component given above. At the same time they generally impose a set of rules concerning the packaging of the component and some times require the implementation of specific interfaces, that will be used by the technology for managing purpose.

Before all two different kinds of component models have been defined. The first, called **desktop components**, provide mechanisms that permit the integration of component deployed on the same system. This is the case of COM and JavaBeans. The second model, referred as **distributed components**, provide mechanisms for integrating components that could be dispersed on more than one physical system. Distributed component technologies obviously rely on middleware technologies at least for what concern the necessary communications among components deployed on different hosts.

The implementation choices made by the different component model technology providers are strongly different and it is not possible in general to take a component from one world (a component model technology) and deploy it into another world. However apart from implementation details one service constitute the basic starting point for defining a component model. This service is referred as **naming and locating service** and its task it to provide, at run time, to the components that need a service, a reference to another component that provide the specified service. Through the implementation of a naming and location service a component model permit the real implementation of software elements that do not contain embedded references to the final providers of the required services.

Several different component models have been defined so far, however three of them currently lead the scenes (COM/.NET [2], CCM/CORBA [3], EJB/J2EE [129]) and are supported by big companies/organizations such as Microsoft, Sun Microsystems and OMG. The technologies providing the support for such component models are really complex stuff. In particular EJB and CCM are a proposal for server side components for which the objective is to make easier and faster the development of complex server side services. Many different services are provided to components by the the applications server implementing such technologies, such as transaction management, life cycle management, security management and many other.

These and other component models are briefly described in [82], in which for each model details concerning, interface implementation mechanisms, packaging, deployment details, and services to the component provided by the different technologies are discussed.

Part II.

Testing in the Component Based Software Arena

In this part I introduce concepts related to software testing with particular emphasis on the Component Based field. In particular I discuss:

- In Chapter 4 the general concepts behind software testing;
- In Chapter 5 the implications for functional software testing execution when CB systems are considered;
- In Chapter 6 the application of testing to performance evaluation of complex software systems.

4. Software Testing

4.1. Software Testing Generic

Testing is a fundamental activity in the development process of a software system. Its importance has been traditionally underestimated and often research results in the area encounter great problems in becoming state of the practice, moreover testing, which is an activity that to be really effective requires the support of specific automatic tools, is often carried on, within software industries, in a manual way (a particular case is that of eXtreme Programming in which testing is considered a "first class" activity in the development of the final code [29]).

Many different types of testing techniques and strategies have been studied and developed, however all of them share the same final objective that is to increase the confidence in the correct functioning of the system when it will be released. To this end testing searches for discrepancy among expected and observed behavior of the system on a finite number of execution. In software testing theory the occurrences of a discrepancy is referred as a **failure**. The originating cause of a failure is instead referred as a **fault**. A fault can reside in the system for a long time before a concrete manifestation of it leads to an observed effect (a failure). The intermediate unstable state of a system before the manifestation of a fault is generally referred to as an **error** [35]. We can restate the above definitions saying that the ultimate goal of testing is to look for software executions that lead to the manifestation of faults. Before going in a deeper technical explanation about concepts related to this discipline, it is useful to remember that, as firstly stated by Dijkstra [75], testing can reveal the presence of faults but can never prove their absence. Another interesting consideration, on the particular nature of software and of the connected difficulties in testing software systems, has been firstly stated by Hamlet [96]. In particular Hamlet refers to the inherent discontinuity of software systems that, differently from other engineering artifacts, hinder the possibility of making inferences on system behavior starting from samples. In other word this means that it is generally impossible to derive the correct behavior of the system on a input, starting from the correct behavior of the system on another test case.

Pfleeger [141] provides a useful classification of different kinds of faults. The proposed list should lead the tester to different choices depending on the type of faults that he/she looks for, trying to create, each time, specific conditions to increase the "probability of manifestation" for the searched fault. In particular in [141] the author proposes the following categories of faults:

- **syntax faults**: occurs when the programming language constructs are not properly used;
- **algorithmic faults**: occurs when a part of the program logic do not provide the proper output for a given input because something is wrong with the processing steps;
- **computation** and **precision faults**: occur when a formula's implementation is wrong or does not compute the result to the required precision;
- **documentation faults**: occurs when the documentation description does not actually match to the program behavior;
- **stress** or **overload faults**: occur when the size of the data structures used in the program are less dimensioned than necessary;
- **capacity** of **boundary faults**: occur when the system's performance become unacceptable as system activity reaches its specified limits;
- **timing** or **coordination faults**: particularly relevant for real-time systems, they occur when the code coordinating these events is inadequate;
- **throughput** or **performance faults**: occur when the system does not perform at the speed indicated by the requirements.

From the list above it descends that testing can involve many different activities related to the verification of a piece of software. In the following of this chapter I mainly focus the discussion on what is generally referred dynamic testing that concerns the evaluation of a piece of software through its execution. We can better characterize this kind of testing giving the following definition firstly proposed by Bertolino in [34]:

> *Software testing consist of the* ***dynamic*** *verification of the behavior of a program on a* ***finite*** *set of test cases, suitably* ***selected*** *from the usually infinite executions domain, against the specified* ***expected*** *behavior.*

In the above definition the author highlighted the words that contribute to establish the main distinctive characteristics of software testing. In particular *dynamic* implies that testing requires the execution of the program on some inputs; *finite* means that the number of execution must be finite even if the complete execution of a complete test would require an infinite number of executions; *selected* refers to the fact that testers should apply rules to select test cases from the infinite possible executions, being aware that different techniques lead to completely different results; *expected* refers to the necessity of having some means for evaluating the result of the test deciding if it is acceptable or not. This problem is generally referred in the literature as the **oracle problem**.

4.2. Verification Techniques and Objectives

Many different techniques have been developed to discover faults. Tester teams should be aware of them and applying the most promising for finding specific faults. In literature the different techniques are generally grouped in two categories:

- **static analysis techniques**: the techniques included in this category do not foresee the execution of the system, instead intend to find faults in the system only manually or automatically scrutinizing the artifact produced (such as code, documentation, software models). Main elements in this category can be considered *code inspection*, *algorithm analysis and tracing* and *formal proof*. In particular different formal techniques can be applied to prove the correctness of an artifact depending on its nature. The most famous approaches proposed are *symbolic execution*, *theorem proving*, *model checking*.

- **dynamic analysis techniques**: this category includes such techniques that foresee the execution of the system to derive information on it. *Testing*, *profiling*, *simulation*, *prototyping*, *timing analysis* and other, are the main elements of this category.

In the previous section I listed a possible classification of faults that should drive the tester to the choice of a specific technique. However another important parameter that must be considered in the definition of a **testing suite** (that is a set of test cases that can be executed to assess a particular characteristic) is the final **objective** of the testing (the property of the system that we want to empirically evaluate). In [38] the authors provide the following list (obviously not complete):

- **Acceptance/qualification testing**: this is the last test action before the deployment of the system in the final application environment. Its main goal is to verify that the software respects the customer's requirements.

- **Installation testing**: this test action is carried out when the system is installed in the final execution environment. As in the previous case the main goal is to assess the conformance to the customer's requirements, but in this case in the final application environment. At the same time it is useful to check that the procedure that must be followed to complete the installation it is correct and lead to a right installation.

- **Alpha testing**: this test activity is performed installing a partial or complete system on an in-house environment and exploring already implemented functionality and business task, before a possible release.

- **Beta testing**: the same as in the previous point but in this case the system is provided to external user that test the system on different environment.

- **Reliability achievement**: in this case test are performed for assessing the behavior of the system for the most likely execution request. In this sense an operational profile must be defined and test cases selected to stimulate corresponding use cases.
- **Conformance testing/Functionality testing**: in this case the test are executed to assess the functional conformance of a system or a piece of system to the specifications.
- **Regression testing**: in this case after a piece of the system is modified/substituted test session will be performed to have a new evaluation of the resulting system.
- **Performance testing**: this is specifically aimed at verifying that the system effectively provides the necessary performance specified.
- **Usability testing**: in this case we need to assess that the system is easily usable for the user, that its functionality are easily understandable, that the documentation is clear and so on.

4.3. Unit, Integration and System Test

We can start this section saying that even in the testing field, modularization gives its useful contribution. In fact it is a standard way, in testing, to proceed with different successive phases of testing, each one focused on a different size of an agglomerate of "pieces" of software. Three different kind of granularity for these agglomerates are generally considered for testing purpose, such as **Unit**, **Integration**, and **System**.

The scope of *unit testing* typically comprises a relatively small executable. Generally the interest is focused on functionality instead than on other characteristics.

Even when a complete test is successfully executed on two or more units, there will be no guarantees on the correct behavior of the system obtained integrating such units. Scope of the *integration testing* is to verify the correct interactions among the aggregated units. Depending on the order used to aggregate the different units and the relations - uses/is used - among these units different strategies can be applied. If the relation among the different units can be organized in a hierarchy we distinguish among **bottom-up integration**, the system is integrated starting from the low level element of the hierarchy, **top-down integration**, in this case the integration start from the element at the top of the hierarchy and necessary **stubs**[1] are provided, **big-bang integration**, the complete system is immediately created after each unit being

[1] these are pieces of code that simulate the behavior of a unit for another unit that need services from the first unit

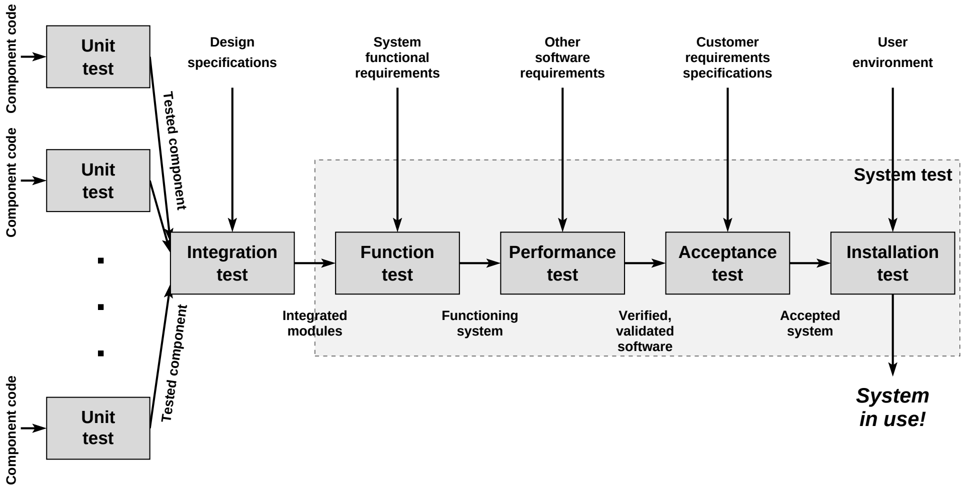

Figure 4.1.: Testing steps (partially from [141])

tested, **sandwich integration**, obtained mixing steps of top-down and bottom-up. For a comparison of the different approaches see [141].

Finally *System testing* focus the attention on the whole system with particular reference to functional requirements, even though different analysis can be performed considering the whole system. Figure 4.1 illustrates how some testing steps can be ordered to derive a final functioning system. As we can see the system test phase is split in more phases depending on the specific objective that is pursued.

4.4. Test Case Selection

Probably the test case selection phase is the most important activity of the testing process, and certainly the one that can make a difference. Several different strategies have been proposed to select test cases from the generally infinite set of the possible executions. Tester teams should define the strategy to apply on the base of the specificity of the system under test and of the available information that can be fruitfully used for testing purpose. Using an analogy from a real life domain Hamlet [97] considers testing similar to fishing, it is generally necessary to make several tries before catching something, further expanding the analogy we can say that to increase your chances of having a good catch you need to know the habits of the specific kind of fish that you want to fish and you have to choose the most suitable fishing techniques. However even when you do not fish anything you cannot say that there is no fish in the sea. Being the testing phase one of the most expensive in software production the analogy suggest the importance of the testing team and of its expertise to find the greatest number of faults within the limited resources.

Given the great number of possible executions, even for a small program, it is of basic importance to derive an automatic procedure for the selection of the test cases when a strategy has been chosen. The procedure should derive a test suite according to the selected **test criterion**. A formal definition of a test criterion is provided by [35]:

> *A test criterion C is a decision predicate on triples (P,RM,T), where P is a program, RM is a reference model related to P, and T is a test suite. When C(P,RM,T) holds, it is said that T satisfies criterion C for P and RM.*

Test criterion is the basic ingredient of those approaches referred as "partition testing". In fact the adoption of a criterion induce a partitioning of the program input into equivalent subdomains, where the equivalence relation is referred to the ability of a test case of discovering a faults. An opposite way to proceed for selecting the tests is called **random testing**, in which the inputs are selected from the infinite set in a random way with no reference to a test criterion. Three main techniques [49], referred as code based, specification based, and fault based, can be used for selecting test cases. The starting point of these techniques are different, the first being applicable only when the source code are available, the others instead foresees the presence of some model related to the system under development. For each one of these categories, several different strategies have been proposed. However it is worth to note that previous researches highlighted that the combination of more than one selection technique produces better results [110] and that different test criteria aid to discover different types of faults [27].

Code-based techniques. The motivation behind the code based testing is that a fault can be discovered only if the parts of code related to the faults are executed. The program is represented as a flow-graph and different criteria of coverage can be adopted. The ideal but unreachable target, since it requires an exponential number of test with respect to the number of conditions in the source code, is to cover all the possible paths along the program flow. Many different coverage strategies have been proposed in literature starting from the statement coverage that require to execute all the possible statement in the code, to the branch coverage, that requires to cover all the possible different branch in the flow, until the already cited path coverage. In figure 4.2 the relative strengths of many different strategies, deeply discussed in [30], are illustrated. Although code based seem a captivating strategy, two major problem in the automatic application of it must be at least cited:

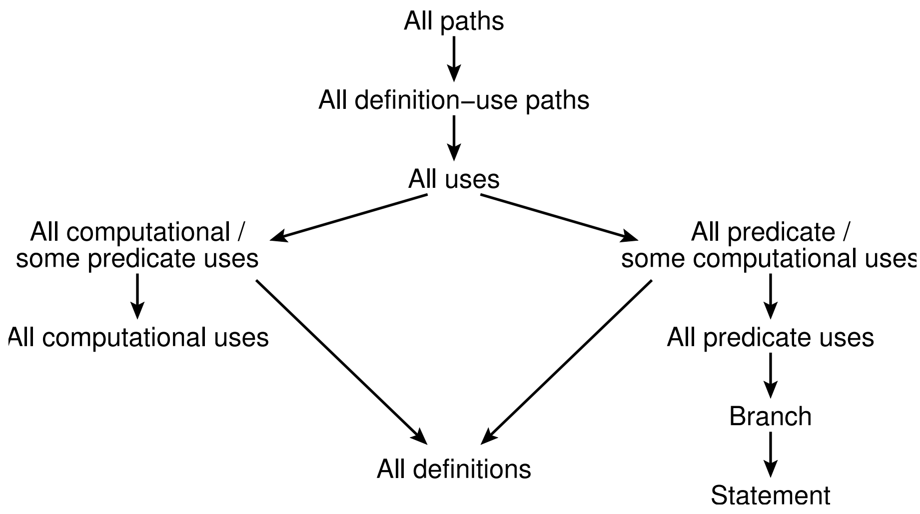

Figure 4.2.: Coverage strategies relations [30]

1. it is possible that the code contains path that can never be reached by an execution (**infeasible path**) so invalidating the effort for searching an execution.

2. finding an execution that cover a particular path has been demonstrated to be an undecidable problem [175], and so a quite difficult problem to solve in practice.

Specification-based techniques. With specification-based strategies the RM in the test criterion is derived form the documentation describing the system. Depending on the nature of the resulting reference model completely different strategies can be applied. The advent of the Unified Modeling Language rejuvenated this topic since diagrams seem the natural instruments for the application of specification based strategies. In particular in [26] the authors propose a tool for selection of test case starting from sequence, class and use cases diagrams. A European project recently terminated (AGEDIS [98]) explored the use of state diagrams for test case selection and execution. However specification based strategies have been initially proposed for software model defined using formal mechanisms such as Z [99] and LTS [162]. Also interesting approaches have been proposed for the selection of test cases starting from an architectural description [132, 37]. Finally considering as a starting model the usage profile of the system the resulting strategy is that generally applied for reliability evaluation. In Chapter 9 an application of this criterion will be presented for performance testing purpose.

Fault-based techniques. Techniques in this area foresee the introduction of faults into the code to verify how many faults will be revealed by a test suite (**mutation testing** [85]) or try to reuse the expertise of programmer in the specific domain to guess where faults are more probably hidden (**error guessing** [134]).

4.5. Test Execution

So far we have mainly presented testing as a theoretic matter that tries to select the best test cases to execute when a precise objective has been defined. In reality also the execution of a test case presents per se challenging questions such as, the launching of a test case, or how to decide that the obtained result of a launched test is correct or not. In particular the former problem concerns how to create the initial conditions that permit the execution of the test and how to monitor the execution of the test to impose that precise interactions take place, the second, instead, refers to the problem of creating a test oracle that can be used to check if the result of the test is correct or not. The latter problem is far from easy and requires, in the general case, the implementation of a system that should provide for each test the correct answer that we expect. Obviously if we can have such a system the oracle system itself can be considered a right implementation of the system that we are trying to develop. Therefore the implementation of the oracle should be much cheaper that that of the system. Generally the oracle provides only approximated solutions and check for necessary conditions. In [176] the author deeply investigate the problems related to the development of a test oracle, instead for a survey of proposed solutions see [25].

4.6. Testing Object-Oriented Software

On the previous survey I did not refer to any particular technology used to implement the system, therefore the considerations made above are generally valid. In this section I want to give some more details on testing object-oriented software illustrating how the specificity of the object-oriented language raises new problem to the testers and create new and specific possibility for **bug hazard**, that is a circumstance that increases the change of a bug. Since in a certain sense the world of components can be considered as an evolution or the extreme consequence of certain object-oriented characteristics, the discussion that follows is relevant also for the component-based community, especially when the reuse is made using object-oriented constructs, such as inheritance.

Graham [93] proposed an interesting classification of the specific features of the Object Orientation for testing purpose. As illustrated in Figure 4.3 she considered in the first group the features that make testing easier when OO languages are used, in

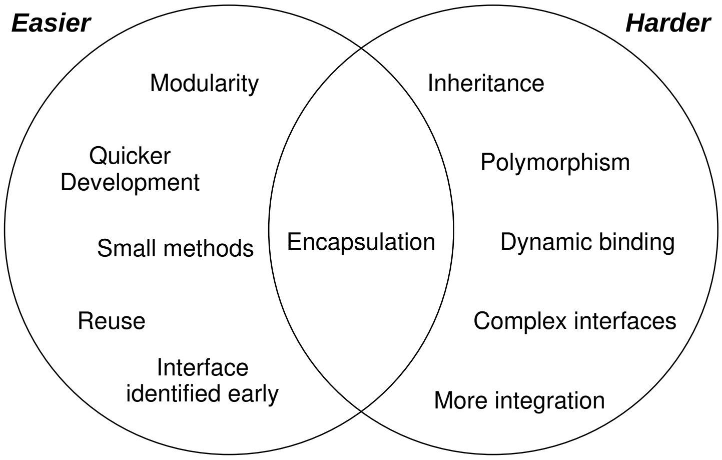

Figure 4.3.: Easier and Harder parts of testing object-oriented systems [93]

the second group she inserted the elements that make testing harder. For instance giving the concept of class the test of the unit became easier since the unit are generally smaller and clearly defined. However nothing comes free since this advantage will be paid during integration testing when the kind of interactions are generally more complex, as a consequence of more complex relations among the objects. Following the indication given by Graham and using a fault-based strategy, as first level selection mechanisms, we should focus the attention, in testing object-oriented systems, to the new features such as **polimorphism**, that is the possibility of redefining methods in subclasses and then the consequent different behavior of the same object (method) in different context, **inheritance**, the possibility of inheriting definitions from the super classes, and at the **interface** and **integration** level.

The characteristic of OO code that probably most hinder the possibility of testing is **Dynamic Binding**. Using this mechanism it is possible to change the object provider of a service in any instant at run time. Moreover in association with polimorphism, dynamic binding permits the definition of the binding to the method providing the service, only when the method is actually invocated. For a comprehensive discussion of OO oriented testing see [49], in which the interesting notion of **testing pattern** is introduced with a strict analogy to the design patterns [86] defined for the design phase of a software system.

In this chapter I have given general information on the testing phase. In the following three chapters I provide a deeper insight on particular aspects of testing considering component based software as the development paradigm. In particular in Chapter

5 I give an overview of the proposed mechanisms for test cases execution when component are considered. Finally in Chapter 6 I discuss the proposed solutions for the empirical evaluation of performance characteristics of complex middleware based software applications.

5. Testing Techniques and Tools for Component Based Software

Although it is generally agreed that the implementation of a system out of components needs specific approaches, clear guidelines and new processes have not yet been established. As for the other development phases, the testing stage as well needs a rethinking to address the peculiar characteristics of CB development [44]. One distinctive feature of CB production is the co-existence all along the development process of several and new stake-holders. A CB process must in fact foresee and manage the spreading, in time and space, of different tasks among several uncoordinated subjects [44]. For testing purposes, in particular, we must distinguish at a very minimum between two subjects, the component developer and the component user. The first needs to test the software component in order to validate its implementation, but cannot make any assumption on the environment in which the component will be employed. The second, instead, needs to (re)test the component as an interacting part of the larger CB system under development. In this respect, an important requirement is that the component user, on the basis of what he/she expects from a searched component, i.e., with reference to the system specification/architecture, can develop a test suite and can then routinely (re-)execute these tests - without too much effort - to evaluate any potential candidate components. The latter is precisely the question that we address here.

A testing campaign by the component user, possibly in combination with usage of formal methods and Design by Contract (DbC) approaches [125], is also recognized as a suitable means to alleviate a new emerging problem in the area of CB production, generally referred to as the *Component Trust Problem* [126]. This problem points at the component user's exigency of means to gain confidence on what a component produced by someone else does and how it behaves. Obviously this issue is especially hard for components built by third parties and for COTS (Component-Off-The-Shelf), which are generally delivered without the source code. However, also in the case of components reused internally to an organization, the difficulties of communication between teams and the lack of a clear documentation can produce to some extent similar effects. Moreover, even though a component has already undergone extensive testing by its developer, since complete testing is clearly impossible and the developer cannot know in advance all the possible application domains or what components will interact with the produced component, some kind of testing against the component user's specifications remains always necessary [173]. In this sense, it is also illusory to hope that reuse drastically diminishes the need for testing [49, 173].

On the foundational side, Rosenblum [145] outlines a new conceptual basis for CB software testing, formalizing the problem of test adequacy (with reference to a particular criterion) for a component released by a developer and deployed by one or more customers. He introduces the complementary notions of *C-adequate-for-P* and of *C-adequate-on-M* for adequate unit testing of a component and adequate integration testing of a CB system, respectively (where C is a criterion, P is a program and M is a component). This work constitutes a starting point for a revision of the theoretical concepts related to testing components, but a lot of work remains to be done.

Instead, concerning the practical aspects of testing from the component user point of view, we cannot talk about *one* generic testing approach. For testing purposes, in fact, components can be classified depending on the information that is carried on with the component itself. In this sense, we could figure out a continuous spectrum of component types, at one extreme of which there are fully documented components, whose source code is accessible (for instance, in the case of in-house reuse of components or open-source components). At the other extreme of the spectrum there are components for which the only available information consists into the signatures of the provided services, which is the typical case of COTS (commercial off-the-shelf) components. Clearly, the testing techniques to be used by the component user will be quite different depending on the type of component. For instance, in the case of COTS, the unavailability of code hinders the possibility of using any of the traditional code based testing techniques.

5.1. CB Development Process and Testing Phase

In this section we discuss some issues related to the definition of a CB development process. It is not our intent (nor it could be done in only one section) to completely unroll this topic. The interested reader can refer to [63, 68]), while an overview of CB life cycle processes embedding quality assurance models can be found in [60]. Our goal is to relate the framework that we have developed to the relevant phases of the development process. Besides this overview provides information useful to better understand how the framework can be used by a hypothetical component customer.

The very idea of producing software systems out of components is older than thirty years [117], but it is only in the last years that strong efforts have been made towards the real affirmation of this new methodology. Today, many component models exist, and from a pure technological point of view it is quite "easy" to build systems by composing components. Technology advances have in fact delivered component models and middleware addressing the questions of composition, interaction and reuse of components. However, a comparable progress has not been done regarding the definition of a process apt to develop component-based systems. Thus, designing a system that will be implemented by "composing components" still remains a challenging activity, and further study on the subject is needed.

Some general considerations concerning such a process and the various activities that compose it can be done. The *implementation phase* mainly deals with the development of what is generally referred to as the "glue" code. This code is the necessary instrument to facilitate the correct interactions among the different components. The components instead are not generally implemented, but are looked for in the in-house repositories or on the market, through what is generally referred to as the *provisioning phase*. After one or more candidate components have been identified, it is necessary to evaluate their behavior when integrated with the other already chosen components. This phase can be referred to as the *selection and validation phase*. Obviously in this phase testing can play an important role. In fact on the basis of the specifications for the searched component, testers can develop useful (functional and architectural) tests to be executed to evaluate each candidate component.

If the above requirements for the implementation phase can seem quite obvious, less clear but perhaps most important considerations must be done for what concerns the *specification phase*. In the new process, the emphasis of this phase must be on the re-usability and interoperability of the elements that will be part of the system. As said in the Chapter 2, an important instrument towards this target has been identified in the *software architecture*. In fact, using the specification mechanisms developed for the software architecture, the structure of a system is explicitly described in terms of components and connectors. In a CB development environment it is important to establish a direct correspondence between the architectural components and the run-time components. In other terms, the components forming the software architecture and the interconnections among them must remain clearly identifiable also dynamically, during execution. This feature, in fact, affects the quality of the system in terms of *reuse*, *replaceability* and then makes easier the management of system *evolution*. All of these features are clearly major targets in a CB development process.

Finally, we make some considerations concerning the *testing phase*. For this activity, what we need to figure out is a testing methodology that can allow for the effective testing of a component by someone who has not developed it, and within an application context that was completely unknown when the component was developed.

Traditionally, the development of complex systems involves three main testing phases as illustrated in Chapter 4. In CB development, the three traditional testing phases have to be reconsidered and extended (see Fig.5.1). The smallest test unit becomes here the component. *Component testing* is performed by the component developer and is aimed at establishing the proper functioning of the component and at early detecting possible failures. The tests established by the developer can rely not only on a complete documentation and knowledge of the component, but also on the availability of the source code, and thus in general pursue some kind of coverage.

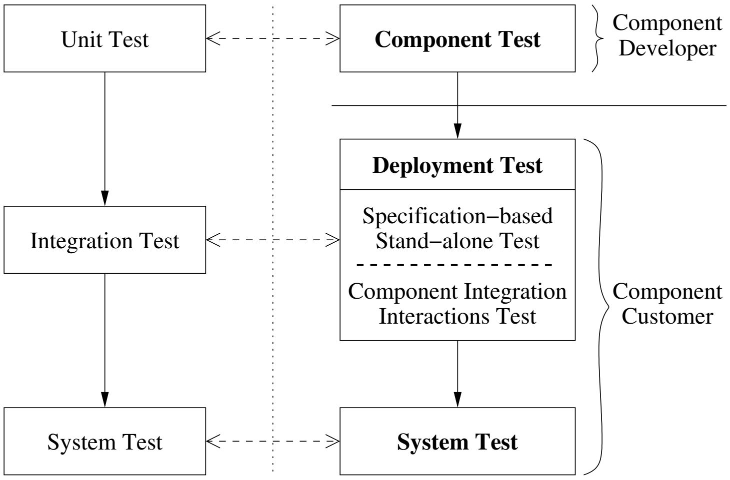

Figure 5.1.: Adapting the test process

However, such testing cannot address the functional correspondence of the component behavior to the specifications of the system in which it will be later assembled. In fact it is not possible for the developer to consider all the environments in which the component could be successively inserted.

The phase of integration testing corresponds to the stage we denote by *deployment testing*, however conceptually the two tasks are very different. Performed by the component customer, the purpose of deployment testing is thus the validation of the implementation of the components that will constitute the final system. In our study we divided this phase in two successive sub-phases. In the first sub-phase the component will be tested as integrated in an environment constituted of stubs that roughly implement the components as foreseen by the specifications. In that manner we check if the component correctly interact with the "ideal" environment. To do this we need a driver that executes the test cases by directly invoking the services provided by the component. In the second sub-phase, we verify the integration between several chosen components. To do this we need means to monitor the interactions among the actual implementations of the architectural components during the execution of some test cases. In this manner we can check whether some wrong interactions among the components occur. Particularly useful to this purpose can be the definition of a "contract" [125] between the provider and the client of a service, as we will better explain in the following. It is worth noting that potential mismatches discovered by the customer during deployment testing are not in general "bugs" in the implementation. Rather they evidence the non conformance between the expected component and the tested one (and hence the need to look for other components).

Also for deployment testing (as usual for integration testing) we can consider to adopt an incremental strategy, allowing for the progressive integration of components into larger subsystems. In the presentation of Component Deployment Testing framework (CDT), that is the specific framework that I developed as part of my Ph.D. (see Chapter 7 for major details), for clarity we speak in terms of a single component, however the framework could be identically applied to the deployment testing of a subsystem (we return on this in Chapter 7), with the introduction of specific mechanisms allowing for monitoring the interactions among its components.

A particular case of deployment testing is when a real component comes equipped with the developer's test suite, and the customer re-executes those tests in his/her environment. These tests guarantee that the "intentions" of the developer are respected in the final environment and their execution generally lead to a more comprehensive evaluation. They can possibly include test cases not relevant for the customer's specific purposes, but that can be however useful to evaluate the behavior of the component under customer's unexpected entries.

Finally, *system test* does not show major conceptual differences with respect to the traditional process (at this level of the analysis) and is performed by the customer when all the various components are integrated and the entire system is ready to run.

To deal with the component trust problem, a focussed, coordinated initiative [126] has been recently launched, acknowledging that the solution cannot be univocal, instead a mix of formal and informal approaches should be applied, including formal validation, Design-by-Contract, testing techniques and others.

In the next section we give a glance to several proposed approaches, in some manner related to testing from the component user's view. The approaches cannot be considered alternative, rather the combined usage of more than one of them can certainly give better results than only selecting one. Obviously the list is not exhaustive, but reflects our comprehension and best knowledge of the literature. We omit the approaches proposed for testing components from the perspective of the component developer, since they address the problem from an opposite perspective (developer view), in which the information available on the components internal structure (such as the source code) permit the use of different, more white-box oriented, methodologies (the interested reader can find useful information on this topic in [122]).

5.2. User Oriented Approaches for Component Testing

Built-in testing approach. The idea behind the "Built-in testing" (BIT) approach is that the component developer can increase the trust of the component user by augmenting the provided component with executable test cases. Running the provided test cases, the component user can thus validate, in the final environment, the hypotheses made by the component developer. To provide test cases with the component, the basic technique [171] is to distinguish between two different "modes" of component execution: the *normal mode* and the *maintenance mode*. In the maintenance mode the component user can invoke particular methods, enclosed with each class constituting the component, that have been added to codify dedicated test cases. Being part of the class, these methods can access every private variable and invoke every method. So the mechanism provides the component user with a powerful means of evaluation, without requiring the component customer to use any specific framework or tool for testing purposes.

The proposed approach though suffers of some drawbacks. The first, and technical in kind, is that the memory required at run-time to instantiate the objects from a class, can become huge and mainly dominated by the need of space to allocate the testing methods; these, obviously, are completely useless in normal mode. The second, and more conceptual problem, concerns the meaningfulness for the component user of developer's defined test cases. As advocated by different authors [173], it is in fact important that the customer develops his/her own test suites so to ascertain that a candidate component be "compliant" with the requirements for the searched component.

Further details on the BIT approach can be found in [24, 95].

Testable Architecture Approach. This approach can be seen as a special case of the previously described approach, and in fact shares the same aims. However, differently from built-in testing, this one prevents the problem concerning the huge amount of memory required at run-time. The idea is to develop a specific testable architecture that allow the component user to easy re-execute the test cases provided by the developer, without the need of enclosing them in the component itself.

In [87], Gao and coauthors require that each component implements for testing purposes a particular interface. This interface has the goal of augmenting the testability of the component. In that manner the developer can then provide the component user with test cases coded in terms of clients that use the testing interface. By foreseeing the presence in the test-oriented interface of methods that use the introspection mechanisms, which are generally provided by component standard models, the same power of the built-in testing approach can be obtained in terms of access to methods and variables otherwise not visible to clients.

Another interesting approach also relying on the definition of a particular framework for component testing has been proposed by Atkinson and Groß[22]. Differently from the previous approach, in this case there is no use of the introspection mechanisms provided by the various component models. As a consequence the framework cannot reach the same power of the built-in testing approach. However this framework is not intended for the execution of generic test cases, but it focuses on providing the customer with specific test cases derived from contract specifications. In order to check the validity of a contract the authors suppose that a component developer implements particular methods for state introspection. In particular these states are defined at a logical level using a component model based on KobrA [83], a modeling tool developed in the area of Product Line (PL) design.

Certification Strategy Approach. The approaches belonging to this class are based on the general observation that the customer of a component is generally suspicious about the information and proof of quality provided by the component developer. Hence, to increase the trust of a customer on a component some authors have proposed different forms of "component certification". A first approach proposes the constitution of independent agencies (or Software Certification Laboratories [167]) for software component evaluation. The main duty of such an agency should be the derivation and verification of the qualities of a component. To this end the agency should extensively test (from a functional and performance point of view) the components and then publish the results of the executed tests and the used environments. However, the inherent difficulties in establishing these agencies suggested that in alternative warranties be derived as the result of extensive operational usage, following some notable Open Source example (e.g. Linux). By coordinating the users of a particular software, a "user-based software certification" could be established [168].

A different approach to certification has been proposed by Morris et al. [58, 130], starting from the remark that using the services of a certification agency could be particularly expensive for a small software company. To overcome this problem a developer's software certification approach is proposed. This approach relies on the specification and release to the component customer of test cases written in a XML format. This format should guarantee a better understanding, on the component customer's side, of the test cases that have been executed by the developer on the component. According to the authors, on the one hand this should increase the trust of the customer on the component behavior. On the other hand, by using suitable tools, it should be possible for the customer to (re-)execute the XML test cases in the target environment. Thanks to this feature, the approach can also be seen as another variant of built-in testing.

Metadata Approach. As said many times, the scarcity of information is the main source of customer suspicion. Then the proposal behind the metadata approach [136] is to augment the component with additional information in order to increase the component customer's analysis capability. Obviously, it is important at the same time to develop suitable tools for easing the management and use of the provided metadata. Different kinds of information can be provided by the developer such as a Finite State Machine (FSM) model of the component, information on pre- and post-conditions for the provided services, regression test suites [137], and so on. Also the proposal of Stafford and Wolf [158], who foresee the provisioning of pathways expressing the potential of an input to affect a particular output, can be re-conducted to this approach. Whaley et al. [177] propose to supply models that express acceptable method calls sequences. In this manner the customers can evaluate their use of the component and check whether not illegal calls are permitted. Finally an aspect-based approach to classify metadata information, so to better support test selection, is suggested in [62].

Customer's Specification-based Testing Approach. To different extents, all of the above approaches relay on some cooperation and good will on the component developer's side: that some specified procedure is followed in producing the component, or that some required information or property about the component behavior and/or structure is provided. However we cannot assume that this is the general case and often components are delivered supplemented of really little information. At this point the unique means, in the hands of the customer, to increase his/her trust on the component behavior remains the execution of test cases that he/she has defined on the basis of the specifications for the searched component. This is the approach that I have developed in Chapter 7. The use of test cases developed on the basis of the component user's specification and in the target environment is useful in any case, but especially when only black-box test techniques can be used. The application of this kind of approach requires, however, the development of suitable tools and methodologies for test case reuse and derivation. It is also of primary importance to develop new means for the derivation of relevant test cases from the specifications.

In our work we consciously make the least restricting assumptions on how the component is developed or packaged, taking the move from the observation that in practice COTS components may still today be delivered with scarce, if any, useful information for testing purposes. Starting from this assumption we developed our framework (see Chapter 7 for details) for easing the execution of test cases derived from the customer architectural specifications.

6. Performance Evaluation of Component Based Software System

Performance is one of the most critical factor in developing software systems. A system that perfectly responds to the functional requirements but fails to meet performance requirements can be certainly considered faulty. Indeed in [174], the authors report that in their researches performance issues account for one of the three major fault categories. Starting from the early nineties performance evaluation has been reconsidered to be an important architectural issue, more than a technical matter, and Software Performance Engineering (SPE) [155] starts to make the first steps to become an important branch of software engineering.

As generally recognized today, in the development of complex software systems it is important to consider performance requirements from the first steps of the development and not as one of the last steps before system release. Real experiences demonstrate that neglecting performance issue in the system architecture development is hazardous, since to fix a performance lack when the system is ready to be run generally leads to the refactoring either of the whole system with generally huge losses of money, or to the use of a not completely satisfactory system.

Performance can be evaluated using two different and orthogonal approaches. The first considers the definition of specific models, abstracting system structure and behavior, that on the base of precise and sound mathematics tools, logically bound to the model, permit to infer the final system performance properties. As we will discuss in the following, the precision of the results provided by this approach are strictly related to the precision of the defined model. We refer to this kind of solutions as **analytical approaches**. Another possibility for system performance evaluation, which can be called **empirical approaches**, is instead the use of test cases that, simulating the use of the final system, exercise some kind of system prototype. It is worth to note that the prototype should reflect as much as possible the final behavior of the system and its development should be greatly cheaper than that of the final system.

6.1. Analytical approaches

The idea at the base of the analytical approaches is to construct a logical model of the system, using tools based on sound and precise formalisms, that permit to infer the performance properties of the systems. Formalisms that are generally used for such objectives are Petri Nets, Markov Chains, and Queueing networks.

Even though the availability of powerful formalisms, that provide reliable predictions on performance properties of the system, is certainly necessary, two major problems are related to the use of them, and effectively hinder their usage. The first problem is referred to the difficulties in representing the system, using an analytical model, starting from the specifications or the final implementation. The second problem, instead, is related to the inherent complexity in the use of such formalisms, which generally require people with specific skills and strong mathematical background; these qualities are not necessarily in the hand luggage of a software designer.

Researchers, to solve the mentioned problems, are today turning their eyes to the emerging, and today definitively affirmed, modeling mechanisms, such as those introduced in Chapter 2. The very basic idea is to augment the mechanisms provided by the different modeling languages with performance connotations. This additional information will successively enable the definition and analysis of a performance model. In this process the general rule is to hide, as much as possible, any technical performance detail from the system designer, that at this point results only indirectly involved in the definition of the performance model for the system.

Several interesting works have been published in the last years in this area and in particular a lot of them focus the discussion to the modeling of complex system following the component-based paradigm. In particular some recent published work:

1. in [140] the authors consider the use of software architecture as the modeling tool. Following the idea sketched above the authors provide a way to attach a performance description to architectural patterns. The performance properties of an architectural pattern is defined using Layered Queueing Network (LQN) [144, 84] that is an extension of the traditional Queueing Network (QN) model. Considering the final architecture as a combination of pattern the author propose to derive the performance model as the combination of the model associated to patterns.

2. in [66] the authors propose an interesting approach, called PRIMA-UML, that introduces an UML based methodology that encompasses the performance validation task as an integrated activity within the development process through the use of performance model derived form early available UML models.

3. in [94] the authors define a language for making predictive analysis of performance requirements for system obtained assembling software components . At

the same time they map the constructs of the defined language on elements of the Real Time UML profile, providing to such elements a precise "performance semantics".

4. in [148] the authors discuss the design and implementation of a *composer*, which assembles library components based on a classification of their declarative XML-based performance description.

5. in [166] the authors propose an approach based on Model Driven Architecture, which foresees a tool that automatically augments a PSM high level model with performance details deriving from the choice of a specific middleware. The resulting PSM models contains enough detail to be directly transformed in a performance models and analyzed using suitable tools.

6. in [178] the authors propose a language called Component-based Modeling Language (CBML), that is based on UML and XML. The language permits to describe performance models of software component and component-based systems. It has the capability to capture the performance-related features of software components, their integration and deployment in the system.

7. in [43, 42] the authors describe a methodology, called Component Based Software Performance Engineering (CB-SPE), and the corresponding tool that permits to infer the performance characteristic of a component based system. The approach proposed is based on the use of the RT-UML diagrams and foresees the composition of performance evaluations made both by the component developer, that annotates the component developed with performance characteristic, and by the component assembler, that inserts component performance characteristics into the model describing for the whole system.

6.2. Empirical approaches

Weyuker and Vokolos reported on the weakness of the published scientific literature on *software performance testing* in [174]. To this date no significant scientific advances have been made on performance testing. Furthermore the set of tools available for software performance testing is fairly limited. The most widely used tools are workload generators and performance profilers that provide support for test execution and debugging, but they do not solve many unclear aspects of the process of performance testing. In particular, researchers and practitioners agree that the most critical performance problems depend on decisions made in the very early stages of the development life cycle, such as architectural choices. Even though iterative and incremental development has been widely promoted [127, 54, 106], the testing techniques developed so far are very much focused on the end of the development process.

Weyuker and Vokolos report on the industrial experience of testing the performance of a distributed telecommunication application at AT&T [174]. They stress that, given the lack of historical data on the usage of the target system, the architecture is key to identify software processes and input parameters (and realistic representative values) that will most significantly influence the performance. The work presented in Chapter 9 extends this consideration to a wider set of distributed applications, i.e., distributed component-based software in general. Moreover, in it we provide a systematic approach to test-definition, implementation and deployment that are not covered in the work of Weyuker and Vokolos.

Performance Testing of Distributed Applications Some authors exploited empirical testing for studying the performance of middleware products. Gorton and Liu compare the performance of six different J2EE-based middleware implementations [92]. They use a benchmark application that stresses the middleware infrastructure, the transaction and directory services and the load balancing mechanisms. The comparison is based on the empirical measurement of throughput per increasing number of clients. Similarly, Avritzer et al. compare the performance of different ORB (Object Request Broker) implementations that adhere to the CORBA Component Model [108]. Liu et al. investigate the suitability of micro-benchmarks, i.e., lightweight test cases focused on specific facilities of the middleware, such as, directory service, transaction management and persistence and security support [111]. This work suggests the suitability of empirical measurement for middleware selection, i.e, for making decisions on which middleware will best satisfy the performance requirements of a distributed application. However, as Liu et al. remark in the conclusions of their paper ([111]), "how to incorporate application-specific behavior in the equations and how far the results can be generalized across different hardware platforms, databases and operating systems, are still open problems."

6.3. Final Considerations

The work presented in Chapter 9 starts from the consideration that nowadays complex component-based software system make use of complex middleware to cooperate. As a consequence we supposed that final system performance should be mainly consequence of execution inside middleware elements, and related to the particular use of the middleware. This hypothesis partly invalidates the use of pure analytical approaches, given the inherent complexity in modeling middleware components. As a consequence the prevision provided by the analytical model are generally rather rough. It is worth noting that even when middleware component could be introduced inside the analytical model it is necessary to refer the specific implementation of the middleware that will be used. In fact, as already highlighted above, different implementations of the same middleware can provide completely different performance results [92].

In this area we think that empirical evaluation could have a good change of success since middleware is generally a "legacy" component that is used for more than one software implementation and therefore it is already present when the architecture is defined. We also note that it is the middleware functionality, such as transaction and persistence services, remote communication primitives and threading policy primitives, that dominates distributed system performance. Current practice, however, rarely applies systematic techniques to evaluate performance characteristics. We argue that evaluation of performance is particularly crucial in early development stages, when important architectural choices are made.

Drawing on these observations, Chapter 9 presents a novel approach to performance testing of distributed applications. We propose to derive application-specific test cases from architecture designs so that the performance of a distributed application can be tested based on the middleware software at early stages of a development process. A first promising result of the application of the methodology is also reported.

Part III.

Proposed Approaches for Component-based Software Systems Testing

In this part I illustrate proposed solutions to the issues presented in the previous part. The result that I report are mainly consequence of fruitful collaborations with other researchers. In particular I introduce:

- In Chapter 7 a framework that make easier the codification and execution of test cases for CB software. This research topic have been explored at ISTI-CNR in collaboration with Antonia Bertolino and has successfully lead to two main publications [46, 142].
- In Chapter 8 an approach for the evaluation of system developed integrating different component, on the base of models derived form the analysis of execution traces. This research topic is still an ongoing work conjunctly explored at ISTI-CNR and University of L'Aquila in collaboration with Antonia Bertolino, Paola Inverardi, and Henry Muccini, and has been shortly illustrated in [47].
- In Chapter 9 an approach for performance evaluation of component based distributed system. This research topic have been mainly explored at University College of London (UCL) in collaboration with Giovanni Denaro and Wolfgang Emmerich and has successfully lead to two main pubblications [73, 74].

7. CDT a Framework for Component Deployment Testing

In view of all the needs depicted in Chapter 5, and considering a scenario in which no information is made available by the component developer in addition to the component interface signatures (the extreme COTS example), we have developed the Component Deployment Testing (CDT) framework. CDT supports the functional testing of a to-be-assembled component with respect to the customer's specifications, which we refer to as *deployment testing*. CDT is both a reference framework for test development and codification, and an environment for executing the tests over a selected candidate component. In addition, CDT can also provide a simple means to enclose with a component the developer's test suite, which can then be easily re-executed by the customer.

The key idea at the basis of the framework is the *complete decoupling* between what concerns deriving and documenting the test specifications and what concerns the execution of the tests over the implementation of the component. Technically, to achieve such a separation, the framework requires the capability of retrieving at run-time information on the component, mainly relative to the methods signature. In other words, the component to be tested must enable introspection mechanisms [103], allowing for the component run-time introspection.

The CDT framework [46] has been designed to suit the needs for component deployment testing, as discussed in Section 5.1. In particular we have focused on developing suitable means for facilitating **customer's specification-based testing**, as above outlined. Our objective is to develop an integrated framework within which the component user can evaluate a component by testing it against its expected behavior within the assembled system. The main features of such a framework should include:

- the possibility of the **early definition of the test cases** by the component user, based on the expected component behavior within the target system;
- an **easy re-use of the defined test cases**, in order to evaluate a set of candidate components;
- the **easy configuration/re-configuration of the assembled system**, in order to execute the test cases;

- the **easy extensibility of the testing framework**, in order to add new features and functionality for testing control purposes;
- the **reduction of the number of test cases** to be re-executed when a component is substituted.

In our approach we distinguish among "virtual", "concrete" and "real" component. The first represents an ideal component fully conforming to the architectural definition. A concrete component is instead a possible implementation of this virtual component, that can be validated through testing. Finally, a real component is a component as it can be retrieved from the market or the in-house repositories. It is generally possible to implement a concrete component by combining more than one real components.

We assume that the component user has identified some requirements for a component derived, for instance, from the system architectural specifications. On the basis of such requirements, (s)he devises a reference component model that we call the "virtual component". With reference to the previous classification, the framework allows for the codification of the virtual component interface in a particular class that will be used as the target of testing (we will expand on this on Section 7.2).

The framework subsumes a component user's development process, according to the considerations expressed in Section 5.1. We intervene at the stage in which the component user is searching a real component matching the virtual component's requirements. The process can be roughly summarized in the following steps:

1. definition of the system architecture, with identification of the components and of the associated provided and required interfaces;
2. coding of the interfaces, as identified in the previous step, into precise interface specifications implemented by the virtual component (these specifications will be successively used as a sort of wrapper for the candidate component);
3. definition of the test cases, starting from the architectural description, to test the defined virtual component;
4. identification of one or more candidate components that could match the virtual component;
5. evaluation of each candidate component on the defined test cases, possibly using stubs if not all services required by them are already implemented.

A main feature of the CDT framework is the decoupling between the specification of the test cases and the actual implementation details of a component candidate for assembly. In other words, our framework allows a component user to start designing the executable test cases before any specific component implementation is available.

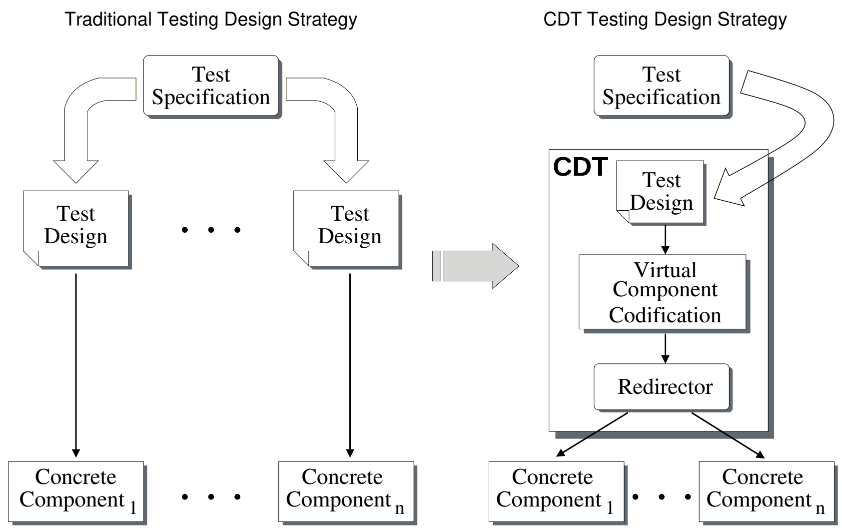

Figure 7.1.: A CDT use example

We illustrate this feature in Figure 7.1. As shown on the left side, following a conventional test strategy, the design of a test case can only start after a concrete component has been identified. Moreover, different test designs can be necessary for each concrete component. Using CDT, instead, the process is simplified, as shown on the right side of Figure 7.1. In fact, CDT makes it possible to perform test design in advance, based on the virtual component specification. Later on, when one or more candidate components are identified, these can be tested by using the framework, without the need to revise the test design. It is the framework that switching from a concrete component to another performs the redirection of the invocations (that a test case always makes to the virtual component) towards the different candidate components. In this manner a clear saving of coding effort can be obtained.

7.1. A Case Study

In this section we present a case study that will be used in the following to illustrate the features of the CDT and how it can be employed during development. It is a proof-of-concept system that has been developed to validate our ideas and to conduct a first experimentation.

Our idea was to develop a CB FTP service system allowing for multiple session clients, i.e., a client can maintain several FTP sessions open at the same time and transfer files between the various servers to which a connection is established. A simple client/server system for FTP service was already available in our laboratory, earlier developed as the result of an undergraduate exam project. However we were completely unaware of the structure of this piece of code. We decided to make an attempt to reuse the real components from this existing system, and to employ the CDT for evaluation.

We started from the specification phase, inspired at large from the process followed in [63]; we identified in the system to develop two main components:

1. a **session manager component**, i.e., a component able of maintaining a reference to all open connections,
2. an **ftp client component**, i.e., a component able of establishing a connection with an ftp server and transferring files via this connection.

For both components, we then defined an interface according to their architectural specifications and the interactions foreseen between them. In particular, for the ftp client component we defined the following interface, with an intuitive meaning for each method:

1. `connect(IPAddress: String, clientPort: Integer, serverPort:Integer)`,
2. `close()`,
3. `list(): String[]`,
4. `download(file:String)`,
5. `multipleDownload(files: String[])`,
6. `upload(file: String)`,
7. `multipleUpload(files:String[])`

Hence, we passed to the implementation phase. We decided to develop in house the session manager component. This was, in fact, a rather specific component. Instead we decided to reuse the ftp client already available to realize the virtual ftp client component. Therefore, we started to analyze the services provided by the real component (the already existing one). To adhere to our hypothetical process allowing for the reuse of black box components, even though we could have had access to the source code, we tried to understand the services that were effectively provided only by analyzing the associated documentation in Javadoc format. The real ftp component implemented the following interface:

1. `open(IP: String, portaC:int, portaS: int)`,

2. `exit()`,
3. `list(): String[]`,
4. `put(file:String)`,
5. `get(file:String):String`

Several differences between the interfaces of the actual and the virtual component can be noticed. From the syntactic point of view, no two methods in the two interfaces had a same signature, and the differences were at various grades. At a first level there were some methods that presumably were performing the same service and differed only in the method's name. A little more difficult were those cases in which the difference concerned the parameters types. Finally, the biggest problem was the absence of methods in the real ftp component allowing for the multiple download/upload service foreseen in the definition of the virtual component. In the following section, we illustrate how using the CDT framework we were able of overcoming these differences and how we defined and used some test cases to validate the implementation.

We notice that CDT is meant for use during development in order to evaluate selected candidate components for reuse within a specified architecture. Once and if the CDT testing is successful, then adequate coding (e.g., wrapping) might be necessary to allow for the effective deployment of the tested component within the system assembly.

7.2. Using CDT

In this section we illustrate how a system developer, who intends to build a complex system by assembling components externally produced, can use the CDT framework for evaluating any possibly retrieved component. To use the framework, essentially the component user must develop for each component three main artifacts, as introduced by the following list:

1. The **"Spy" classes**: constitute the fully conforming codification, in some programming language, of the services provided by the "virtual component".
2. The **test cases**: must be developed to validate a possible implementation of a virtual component. They refer to the virtual component interface referred to in the previous point.
3. The **XMLAdapter**: is a file in which the user must establish some rules expressing how a virtual service is provided at run-time by a concrete component.

```
public class SpyFTPClient extends InformationSwap {

  public void connect(String IPAddr, Integer cPort, Integer sPort) {
    Object[] parameters = new Object[] {IPAddr,cPort,sPort};
    Object ris = driver.executeMethod("connect", parameters);
  }

  /* All the other methods foreseen in this interface are not shown
   * since they share a similar structure with the connect method
   */

}
```

Figure 7.2: Extract of the virtual component codification for the *ftp client component*

In the following of this section we provide more details of how and when these artifacts can be built, with reference to the presented case study.

Clearly such artifacts are not conceived to be used during normal operational usage due to the high performance overhead caused by the reliance on introspection mechanisms. However they can be usefully employed for the development of application prototypes as cooperative glue code.

"Virtual Components" and "Spy" Classes. The "Spy" classes play a key role to facilitate the decoupling between the virtual components and the real components. In short, we foresee that once an architectural component is specified, sufficient information are available to codify a plausible interface (as that illustrated for instance in the case study section). Using this interface as a reference, the component user can thus implement a "virtual component", that has the same interface and can be invoked by the test cases. This "virtual component" is implemented in a "Spy" class. The implementation of the methods of "Spy" has not to be very elaborate, in fact, the methods act only as a form of delegation towards the CDT "Driver" component instance (this component will be presented in Section 7.3). Considering the case study of the previous section, we can codify the required component interface as shown in Table 7.2.

Therefore, the "Spy" represents an interface of the services necessary in order to launch the test cases. But, as illustrated in the following, the association between the services provided by the "Spy" and a real implementation will be established only at run-time.

Besides, we also introduced, at this level, mechanisms for the specification, and for the run-time checking, of contracts. The approach to the specification of provided services using contracts has emerged first in the area of Object-Oriented programming [125]. However, with the advent of the CB paradigm the emphasis on this approach has grown: in fact, contracts have been recognized as a valuable means to transfer information concerning the service behavior between the component developer and component user.

Considering the good potential of this approach, we decided to foresee that the system architect could specify contracts for the services of the virtual components, as an additional means to validate the choice of a concrete component. To do this, the component user should annotate the "Spy" with the definition of the expected behavior for the component in the form of a contract. The syntax to specify the contracts depends from the specific components (as explained in the next section). Basically, contracts can be enclosed in the codification of the "Spy" and they will be checked at run time for each invocation of the "Spy" methods.

Developing the Test Cases. After the "Spy" has been implemented, the testing teams can develop the deployment test cases taking as a reference the "virtual component" codification. Each test case is coded as a method that belongs to a class, that in turn is part of a suite, according to the JUnit [10] conventions. The latter is a well-known Open-Source test tool developed and used inside the eXtreme Programming (XP) community. The goal of JUnit is to provide the developers with a means for the easy codification and execution of the test cases concurrently with development, following the motto "*code a little, test a little, code a little, test a little...*". JUnit fixes some conventions for the naming and organization of the test cases. Two main reasons convinced us to follow the same syntactic conventions (with some minor exception) established by JUnit. The first reason is reducing the learning effort required to the testers (already knowledgeable with JUnit) to use the CDT framework. The second reason, instead, is related to reducing the effort of developing our framework, as we could directly integrate part of JUnit in it. Moreover, by reusing JUnit we have provided CDT with the capability of organizing the test cases in "*target equivalence classes*". Thanks to this feature, when we need to modify a virtual component substituting some internal parts, we can reduce the number of test cases that on average must be re-executed. Obviously it is necessary to apply regression test selection mechanisms to identify the test cases that it is necessary to re-execute.

Due to the reuse of JUnit, it is very easy to add a new test case to a test suite. This is obviously an important factor since new test cases can always be derived as development proceeds. At the end of the test cases elicitation stage, the test cases are in fact packed in a file Jar containing also the corresponding "Spy". Then, to add a new test class, it is sufficient to identify the file Jar containing the specific suite.

With reference to the FTP case study, in Table 7.3 we report an example of a test case that checks the behavior of the component for what concerns the overriding of a file (i.e., when a file with the same name of that just downloaded is found in the directory).

We are also introducing mechanisms for tracing of the execution of the test cases [45]. We foresee to use wrappers for the real components to intercept all the invoca-

```
public void testNoOverwrite() {
  java.io.File file = new java.io.File(serverDirectory + forDownload);
  ((SpyFTPClient)spy).download(serverDirectory + forDownload);
  long lastModified_1 = file.lastModified();

  ((SpyFTPClient)spy).download(serverDirectory + forDownload);
  long lastModified_2 = file.lastModified();

  assertTrue(lastModified_1 == last_Modified_2);
}
```

Figure 7.3: A possible test-case codification

tions. For each invocation the wrapper records all the parameters and return values to reuse them when the component will be modified and/or substituted.

Drawing up the XMLAdapter. As said, a "Spy" is not a real component and its purpose is only to codify the desired functionality for a component, so to permit the early establishment of test cases. In parallel, on the basis of the same specifications, a searching team can start searching the internal repository or the market for components that match the virtual components. We think it reasonable to only focus at this stage on the behavioral aspects of the specification. In other words, we search for components that (seem to) provide the desired functionality, neglecting in this search possible syntactic differences. As a result, several differences at various levels can exist between the virtual component (codified in a "Spy") and the found candidate instances of it.

Now we clarify which is the second artifact that the system developer needs to build for using CDT: the XMLAdapter serves the purpose of specifying the correspondence among real and virtual components, to overcome the possible differences among them, so to permit the execution of the specified test cases. To establish this correspondence, a customer can rely on his/her intuition of what the methods of the candidate components likely do. This intuition can be based on the signatures of the methods, and on any additional documentation possibly accompanying the components, such as the Javadoc description of each method. Obviously this process (which is always the most delicate part of a CB development) is subject to misinterpretation (especially if the components are not adequately documented), and candidate component could behave actually differently from what is expected. However, deployment test execution should highlight such misunderstandings.

An ad-hoc XMLAdapter must be drawn up every time a possible implementation for the virtual component has been identified. We defined a Document Type Definition (DTD) scheme that specifies the legal building box of the XMLAdapter. The information that must be specified in this file can be classified in two different categories, that are reflected also in the structure of the file. In the first part the component user must specify which are the candidate real components that we intend to test. In the current implementation it is possible to specify the real components using the name of a package that contains the component. Alternatively, it is also possible to specify a name registered in a naming service registry. In the first part the user must specify also which are the test cases to be executed on the candidate implementation. They are simply identified specifying the name of the package and of the class containing them. The necessary test cases will be retrieved from one of the file Jars created by the the testing teams by packaging the test cases and the corresponding "Spy". The second part of the XMLAdapter contains information that specifies, for each method in the virtual component codification, how it can be implemented by the methods in the real candidate components. In other words, this part allows for the adaptation of the virtual interface to the real provided service. In particular we have analyzed the following levels of possible differences between these two types of components:

1. differences in the methods names and signatures:
    a. the methods have different names;
    b. the methods have the same number and types of parameters, but they are declared in different order;
    c. the parameters have different types, but we can make them compatible, through suitable transformations. It can be also necessary to set some default parameters;
2. one method in the "Spy" class corresponds to the execution of more than one method in the real implementation of the component.

Obviously the instances listed above are not mutually exclusive, for instance it is possible to have different name methods with different signatures. It may be worth noticing that the symmetric case to 2 (more methods in the "Spy" correspond to one method in the real implementation) is not generally relevant. In fact, the "Spy" is kept simple and typically contains a minimal number of necessary methods. If, say, two methods in the "Spy" correspond to one method in the real implementation, then either the real implementation is not compatible, or we do not need to invoke the two methods alone, but always together and in the same sequence. If so, then it would be more intuitive to indicate only one method in the specification of the "Spy". We imagine that generally the differences between a virtual component and a candidate implementation are generally not so big. We think in fact that big differences in the interfaces can hardly lead to the adoption of the candidate component. Nevertheless, even though it is in principle possible to have a correspondence of type one-to-many in the implementation of a virtual component, we think that in a well established development process, an architecture developed by experts should generally lead to a one-to-one correspondence between virtual and real candidate components.

```
<?xml version="1.0" ?>
<!DOCTYPE Matahari>
<matahari>
  <test_package name="it.cnr.isti.test.components" />
  <test_class name="ClientFTPTest" />
  <real_package name="it.cnr.isti.component.clientFTP" />
  <create_object class="REAL_PACKAGE.Client" object_name="client" />
  <virtual_method name="connect" parameters="serverName portaClient portaServer">
    <exec_method object="portaClient" name="intValue" put_result_in="clientP" />
    <exec_method object="portaServer" name="intValue" put_result_in="serverP" />
    <exec_method object="client" name="open">
      <parameter value="serverName" />
      <parameter type="int" value="clientP" />
      <parameter type="int" value="serverP" />
    </exec_method>
  </virtual_method>
  <virtual_method name="close">
    <exec_method object="client" name="exit" />
  </virtual_method>
  <virtual_method name="download" parameters="file">
    <exec_method object="client" name="get">
     <parameter value="file" />
    </exec_method>
  </virtual_method>
  <virtual_method name="upload" parameters="file">
    <exec_method object="client" name="put">
      <parameter value="file" />
    </exec_method>
  </virtual_method>
  <virtual_method name="multipleDownload" parameters="files">
    <recover_field object="files" field="length" put_value_in="length" />
    <for counter="i" from="0" to="length-1">
      <exec_method object="client" name="get">
        <parameter value="files[i]" />
      </exec_method>
    </for>
  </virtual_method>
  <virtual_method name="multipleUpload" parameters="files">
    <recover_field object="files" field="length" put_value_in="length" />
    <for counter="i" from="0" to="length-1">
      <exec_method object="client" name="put">
        <parameter value="files[i]" />
      </exec_method>
    </for>
  </virtual_method>
  <virtual_method name="list">
    <exec_method object="client" name="list" put_result_in="output" />
  </virtual_method>
</matahari>
```

Figure 7.4: Example of XMLAdapter related to the case study

As a final consideration, the drawing up of the XMLAdapter is certainly not an easy task. However this task can be partially automated and alleviated with the implementation of suitable tools and graphic interfaces.

In Table 7.4 we reported an excerpt of the XMLAdapter for the case study illustrated in the previous section. In this file we reported all the information that permit to an instance of the "Driver" component (see next section) to create the environment for the test phase. In particular we solved all the differences between the two interfaces in this file. As interesting result we discovered, executing the test case in Table 7.3, was that the component overwrote any file with the same name of the file that was currently downloaded. Since we required a more careful behavior we needed to implement a "patch" to this component.

The Graphical User Interface To make easier the use of the framework we are planning the development of a graphical interface that aids as much as possible the user of the CDT framework. Nowadays the developed interface can interact with the user only to start the test phase. We are studying possible extensions for the semi-automatic drawing up of the XMLAdapter, when the candidate implementation of a component has been identified. We briefly describe the features of the current implementation of the CDT interface as shown in Figure 7.5.

Since the interface provides, at this time, functionalities only for the test execution phase, it subsumes that the previous steps, such as "Spy" codification, test case codification and XMLAdapter drawing up, have been already performed. Then as first steps the user must choose an XMLAdapter, drawn up by a searching team, that specifies the virtual component that the user intends to test and the test cases that it is necessary to execute. After the choice of the file the user can choose the run test option (the two check boxes in the upper part of the interface in Figure 7.5 provide this choice). A first possible use case is to run the application only in order to instrument a "Spy" definition with contract specifications. In such a case the framework will retrieve the file Jar containing the test cases and the virtual component codification and create another Jar file that contains the same test cases and an instrumented version of the "Spy". A second possible use case is to execute a testing session. In this case the application will "create" the system architecture instantiating the real components specified in the XMLAdapter, and then the specified test cases are retrieved and executed. It is also possible to execute the two tasks in sequence.

In the lower part of the interface the user, when the test phase execution mode has been chosen, can identify the test cases that, for some reasons, did not finish correctly. In particular we distinguish between the two different cases of "failed test" (the obtained result does not match with the expected one) and "interrupted test" (as a consequence of the raising of an exception). In particular the test cases interrupted because of the violation of a contract belong to the second set.

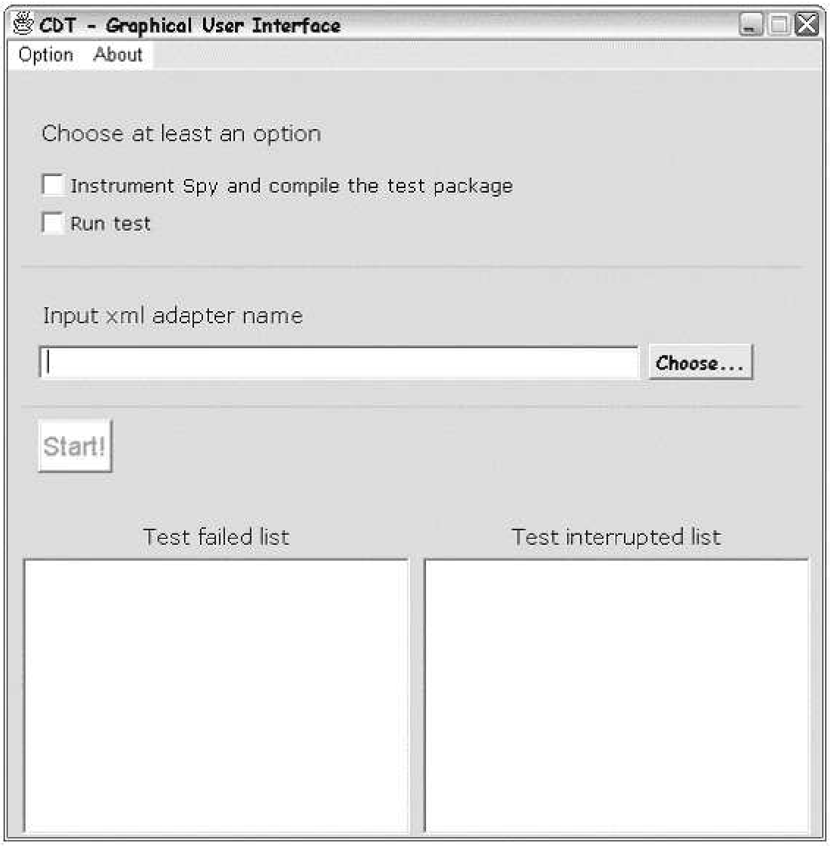

Figure 7.5.: The CDT framework interface

7.3. Framework Architecture

Although the development of the CDT framework, as a proof-of-concept tool, did not follow a-priori a CB paradigm, at the end of the process we strongly revised the code to make it compliant with an a-posteriori developed architecture. In fact, we were conscious of the actual advantages of having a precise correspondence of the code to a high level architecture. Therefore the revision was aimed at isolating as much as possible functionalities belonging to different architectural components, and we created a different Jar file for each one of them. What we would like to have was a modifiable, flexible and extendible implementation. To this purpose we have used the `interface` construct of the Java language and strongly limited direct references

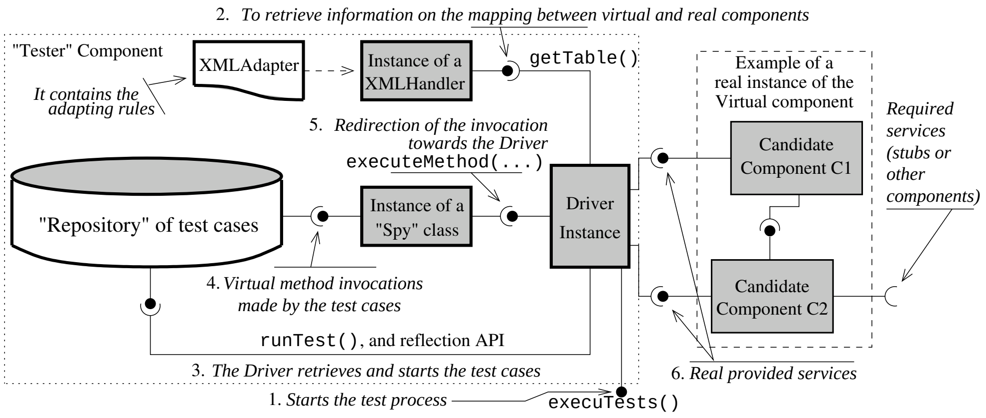

Figure 7.6.: The Structure of the CDT framework

among classes belonging to different Jar files (components).

In the rest of this section we describe the main elements (components) of the CDT architecture and how they interact in order to test a software component, or a component-based sub-system. We have identified the following five main elements in the architecture of the CDT framework:

1. Repository component
2. Contract instrumenting components,
3. Test execution driver component,
4. XML parser component,
5. Interface and coordinator components.

In Figure 7.6 it is shown how, at run time, instances of the main elements in the approach, that will be described in the following, cooperate in order to test a real instance of a virtual component.

Repository Component. We have seen in the previous paragraph how a generic user must codify, following the JUnit test "template", the test cases that will be successively used to test a real implementation of a virtual component. Correspondingly we need to foresee in the system architecture, a repository component which should provide suitable services for the storage of a specified test suite, and suitable services for the retrieving of one of them when its name is specified. In our implementation we have chosen to pack a test suite and its corresponding "Spy" class in a Jar file. At this point we can identify the test suite by the name of this file. Therefore, in the current implementation, the component providing the repository service is abstracted by the Java API for file management, and it is actually implemented by the file system running on the machine. The services provided by the file system are used by two other components

1. the component that can be used to instrument the "Spy" class with contract definition,
2. the component that need to retrieve a test suite to start a test phase

Contract Instrumenting Component. As seen in the previous section we allow testers to insert contract specifications in the definition of the methods of a "Spy". Obviously in the case of the virtual component definition the specification of a contract can only contain reference to the methods foreseen in the definition of the "Spy". No access to a feature of a real implementation of the component, as a field, can be done. We chose to integrate an externally developed component for the management of contracts, and all technical details concerning contract definition (e.g. source instrumentation using special tag in Javadoc comments) reflect the requirements of the integrated component. There are several alternatives to introduce contract checking in the Java language and we choose to integrate **iContract**, a free component (not open source) that can be downloaded from the Internet [7].

iContract works as a preprocessor and generates, from an instrumented Java source file, a source and a class file in which each method starts and finishes with contract checking invocations. The tag that must be used to define a contract are "invariant", "post" and "pre", that will be followed by strings conforming to the following format:

$$<\text{ContractExpression}>[\#\text{ExceptionClassName}]$$

in which <ContractExpression> it is a boolean expression that codes the contract, and [#ExceptionClassName] is the name of an exception that must be raised if the boolean expression is evaluated to *false* (if no exception is declared a `RuntimeException` is launched). A contract can be defined using any legal boolean expression following the Java syntax. Moreover iContract provides the following operators that can be used to define more complex contract expressions:

1. **forall**: this operator permits to specify a rule that must be true for all the elements in a set,
2. **exists**: this operator is evaluated to true if the expression following it is true at least on one element in a set,
3. **implies**: an expression in the form "I implies C" means "if I then check C". Then if the expression I is true also C must be evaluated.

The *contract instrumenting component* interacts with the *Repository component* in order to retrieve "Spy" classes that must be instrumented, and to store the instrumented version of the retrieved virtual component definition.

Test Execution Driver Component. It constitutes the core of the approach. Main duties of this component are the correct setting of the test environment and the redirection of the invocations made on the virtual component to the real implementation of it. To set up the test environment the component asks to the *XML parser component* information concerning the correspondence among the virtual component and the real components. The real components can be provided as packages and classes that must then be instantiated by a specified constructor or simply by names registered in the registry used by a naming service.

Two different services are provided on an instance of this component:

1. `execuTests()`: the invocation of this method starts the deployment test. Hence as first thing the Driver retrieves the information reported in the "XMLAdapter" via an instance of the XML parser component, and acts as described above. At this point it needs to retrieve the test cases to be executed. Hence, the Driver, using the service provided by the repository component, retrieves the Jar file containing the test cases corresponding to the virtual component under test. The tests to execute can also be a subset of the tests contained in the package, and the choice is made following the JUnit rules. After having identified the tests and the "Spy" the Driver instance sets itself as the target for the invocations of the retrieved "Spy" class. At this point the test phase can be started with the execution of the first test case.

2. `executeMethod (String name Object[] par)`: this service is invoked by the instance of the "Spy" class that has been created by the invocation to the method described above. The aim of this invocation is "to inform" the Driver instance of the method invoked by the test case. On the basis of this information and of the data retrieved from the "XMLAdapter", the Driver can decide the corresponding method/methods in the real implementation of the component to invoke.

From the description it is clear that this component is at the center of the system architecture, from where it interacts with the *repository component* to retrieve the test cases, with the *XML parser component* to retrieve information concerning the set up of the system under test. At the same time the Driver is started by an instance of the *interface and coordinator component.*

XML Parser Component. This component provides a parser to handle the XML file that we defined to define the information concerning the "test plan". Its services are accessed by the *Driver component* in order to retrieve information for setting up the test environment.

Interface and Coordinator Components. This component implements the functionalities for the interactions with the user. We have already described the elements of the interface that can be used by the user to instrument a "Spy" class or to execute a test session. After that the information on the task to execute have been inserted in the graphical interface this component creates an instance of the Driver component and starts the execution of the test phase.

7.4. Conclusions and Future Work

In this chapter I presented a framework for the easier and efficient execution of test cases in a component based environment. The framework is meant to give a partial answer to the need of new techniques for test derivation and execution since the traditional ones have been recognized as inadequate. In particular the main intention of the framework is to provide a simple mechanism for the execution of test cases derived by the component user on the basis of the system architecture specification, to validate the choice of possible candidate components. We intend to release shortly, for free download, a beta version of the framework implementing the illustrated functionality. We verified our ideas on a simple case study that has been used in this chapter to present how a generic component user can take advantage from the usage of the framework. In the next future we will further investigate the advantages that the use of the framework can bring, and develop add-on tools to aid the user of the framework. In summary, the main advantages that we have experimented using the framework include:

- Decoupling of test specification and test design from the components implementation;
- No ad hoc requirements imposed on the candidate components for testing purposes;
- Easy reuse of test cases;
- Test suite flexibility (it is easier to add new test cases to a suite);
- Simple mechanisms to group test cases for regression testing purposes;

Currently we are also focusing on the definition of new methodologies for the derivation of test cases. In particular we conducted a first investigation on the use of test cases, derived from a system architectural description, for functional component validation [40]. We are also planning an integration of the framework with new methodologies for the non functional analysis, in particular with reference to performance analysis [41].

8. An Anti-Model-Based Approach for Component Software

In the previous chapter I presented a framework that has been developed to make easier, and better shaped to the CB software needs, the execution of test cases. The framework does not make any assumption on how test cases have been selected and on how the results of the executions of them should be used for deriving particular properties for the system under test. Objective of this chapter is to introduce a novel approach, that I started to investigate in collaboration with Antonia Bertolino from ISTI/CNR, Paola Inverardi and Henry Muccini from the Computer Science Department of the University of L'Aquila, for the testing based analysis of software systems developed assembling software components. The work is still an on going work and certainly more experimental results must be collected and analyzed to better understand the actual relevance of the approach for the evaluation of CB systems. Nevertheless the results obtained so far and the approach interesting promises, induced me to thinking that it is worth to introduce and shortly discuss the approach even in this PhD thesis.

Even if we agree with the usefulness of model-based testing, there can be several reasons why such an approach *cannot be applied* or is *too expensive* for deployment in a specific context. One generic barrier to a wide adoption of model-based testing is its *inherent complexity*, which requires a deep expertise in formal methods, even where tool support is available – as testified in the AGEDIS project [98]. Another obstacle is the difficulty in *forcing the implementation to take a defined path as identified in the model derived test sequences*. The latter are generally expressed at an abstract level, while the executable test cases must be more concrete and more informative (e.g., [36]). Finally, one more counter-motivation to the practice of model-based testing can be the use of *legacy systems* or *COTS*, for which behavior models are not available.

Considering in particular *component-based software development*, a system is generally obtained by assembling already existing components, for which we cannot a-priori assume that a specification or the source code are available. In such cases, model-based testing is not applicable, or would be too costly. We assume in fact that the system assembler has a high-level specification of the global architecture, but can only pose in practice very basic requirements on the behavior of the acquired components.

This is the rationale for an **"anti-model-based testing approach"** as the one I outline in this chapter. While model-based testing starts from an a-priori established model and tries to execute some sequences derived from this model, in "anti-model-based" testing we take the opposite direction. We execute the implementation on some sample executions, and by observing the traces of execution we try to infer/synthesize a-posteriori an abstract model of the system.

8.1. The Approach

The objective of our research is to develop a tool that should assist the system developer/assembler in the evaluation of a component based software system. In particular the analysis proposed by the approach is an "after composition" technique that requires,to be applicable, the complete integration of the system under test and the successive collection of specific data derived from test cases execution. Opportunely collected data will be the starting point for the derivation of a system behavioral model successively used for analysis purpose. From a first glance the approach could seem similar to other approaches proposed in the area of reverse engineering. In particular in [56] the authors propose the use of monitoring mechanisms to derive real execution paths that will be successively transformed in UML sequence diagrams. Despite this work partially shares a similar objective with respect to that proposed in this chapter, to be applicable it foresees the instrumentation of the system source code with suitable mechanisms that permit the collection of necessary information for deriving execution traces and so UML sequence diagrams. In our research we turned our eyes to component-based software systems, that invalidate the possibility of instrumenting the source code requiring the adoption of different mechanisms based on component wrapping. Moreover in [56] the analysis step terminates with the derivation and analysis of the sequence diagrams, instead in our approach we are interested in the derivation of a more complete system behavioral model to which analysis techniques such as model checking [64] can be applied. The use of tracing mechanisms for component-based verification has been also proposed by Mariani in [116]. However the goal of his approach and of the consequent collection of execution traces, is not to derive a system model for analysis purpose, but he uses traces as a sort of oracles for regression testing purpose, applicable when one of the components in the system is substituted.

Figure 8.1 summarizes the approach we are working on, and as the picture illustrates it is based on four main steps:

1. Derivation of the usage profile for the assembled system, based on a high-level specification of the global architecture;
2. Launch of the test cases derived in the previous steps and monitoring of the system to excerpt execution traces

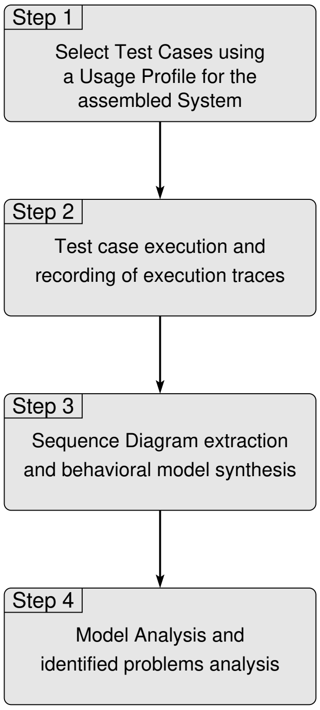

Figure 8.1.: Approach activities

3. Reverse-engineering of the set of sequence diagrams, in order to synthesize a behavior model.

4. Analysis of the model to find possible flaw and further investigation in the case of positive search.

In detail, when a software system has to be produced through assembly of components, wanted system requirements needs to be identified and specified. Whenever the main system requirements are elicited, we may start identifying the architectural components which may reasonably implement the system. We may thus buy the components and create the glue code as a way to produce the desired system.

8.1.1. Step 1: Test Case Selection

In Step 1, suitable test cases have to be identified. As the basic assumption of this approach is that a model for the components is not available, we use the only information that is anyhow available (it may be in various forms): the expected Input/Output functions of the components. This information has to be available in some form, otherwise we could not even use/search for the components. In other words, as a very minimum the component user must know how to solicit the component and what to expect as a reaction. To make such an approach systematic, we will stimulate the component interactions by trying to reproduce the operational usage for the system under test as proposed by the Musa in his works on usage profiles [133].

8.1.2. Step 2: Test Execution and Trace Recording

In Step 2, we have to launch the test cases and monitor the execution traces. Goal of this step is to stimulate the system with inputs, capturing information on execution traces. The idea of capturing traces from code execution is not new. In particular, many strategies aimed to reverse-engineer dynamic models are reported in the literature, many of them surveyed and compared in [56]. The general idea is to instrument the source code, adding some monitors, and run it with some inputs. The monitors help collecting relevant information on run-time execution, such as methods execution, classes and/or objects communication, control or data flow information.

What makes the difference between our monitoring activity and many others is the assumption components are black-box and a component specification is missing. This assumption strongly impacts the way in which monitoring can be performed. In our context, information is gained by instrumenting the glue code used to assemble the components. The information we wish to collect regards the integration between components requiring or providing services. Therefore the tracing mechanism that we need should be able of recording each invocation made by one component on another component. This could be easily obtained trough the use of specific wrappers used to trace either the incoming calls and the outgoing calls for each component. However the tracing task become particularly hard when concurrency, that is the normal case for real component based system, is considered. If concurrency is introduced the tools cited above for tracing executions are not anymore adequate. We need instruments to record the "horizontal" execution of a focus of control inside a component. In other word we need to put in relation the incoming invocations to a component with the respective outgoing invocations. When more foci of control are present at the same time this is not an easy task. The best way to address the problem is the use of an "instrumented run-time" that permits to observes the behavior of the system during the execution and that in particular provides an horizontal view of each focus of control. We started some experiments with Java based components using an old version of the JVM [1] that provides suitable mechanisms for tracing purposes [16].

[1] this has been a major problem

8.1.3. Step 3: Model Derivation and Analysis

In Step 3, the execution traces collected in the previous step are used to synthesize a behavioral model. This one is the most interesting aspect of this research work. In fact, in order to synthesize state machines from execution traces, our idea is to extract scenarios from the execution traces and eventually use such scenarios to synthesize state machines, reusing existing synthesis algorithms (e.g., [17, 164]). In particular this problem can be reconducted to that studied in formal languages and generally referred as **grammar inference** problem [138]. In fact a sequence of invocations can be considered as a particular production of the automata constituted, in the specific case, by the system under test. Therefore the problem is to derive a FSM that can produce the words (scenarios) observed during the execution of the system. An execution trace may be considered as the interleaving of different *scenarios* (as depicted in Figure 8.2), that successively can be integrated with other traces to derive a behavioral system model as illustrated in the next step. To derive a useful model that infer other system behavior besides those actually observed it is generally necessary to provide to the synthesis algorithms rules for combining scenarios. For instance in [164] the authors consider the introduction of high Message Sequence Charts (hMSC) in which the different scenarios are organized in a tree structure. That information can be retrieved from the monitoring step or provided by the developer.

8.1.4. Step 4: Model Analysis

When a FSM, abstractly representing the system, has been derived we use such model to infer properties on the assembled system. This step is certainly, together the previous one, the most challenging in the process. The techniques that we want to apply are those that follow the model checking methodology. For instance we would be interested in verifying that in the produced model there is no deadlock situation. Particularly critical for this steps is the possible presence of implied scenarios [163], that are scenarios not really present in the system but only on the model, which have been introduced as consequence of the inevitable incompleteness of the definition (traces) used to derive the model and of the rules used to combine scenarios. Therefore when the model checking tool finds a counterexample that invalidates the properties that we were checking it will be necessary to extend the investigations on the nature of such counterexample in particular verifying that the scenario actually is present in the real system. The best tool to conduct this steps is probably the derivation of further critical test cases.

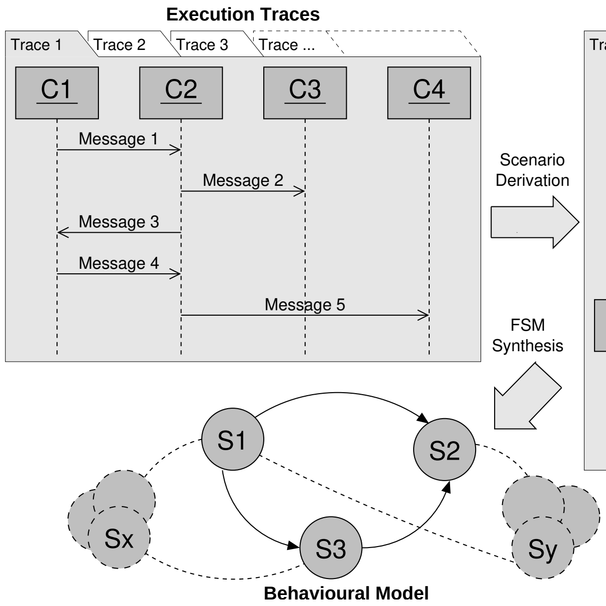

Figure 8.2.: From execution traces to scenarios to behavioral model

8.2. Conclusions and Future Work

Concluding, model-driven specifications have been recently utilized by software engineers for analysis and testing purposes with unobjectionable results. Unfortunately, such analysis techniques cannot be applied whenever the system model is unavailable.

Goal of this work is to propose some initial attempts in this direction; even when system models and software code are unavailable, we outlined how an anti-model-based testing technique may produce relevant results.

In this chapter I simply illustrated how a reverse-engineered model may be produced by analyzing execution traces. However, in future work we desire to investigate how such reverse engineering process may help to discover unexpected behaviors. In particular, our future work will be directed to evaluate, through model checking techniques, how much the implementation is good with respect to expected qualities. Moreover, we may analyze if the system specification produced contains unexpected behaviors. If it does, we may gain some information on how good the selected test cases are.

Currently we have correctly identified the problem and we have addressed some important questions for the real application of the approach. We started also some experiments using a software that simulate the behavior of a set of elevators, developed using the C2 architectural language ([1]) and a framework that provide a simple direct way of implementing a C2 architecture in the Java language.

9. Early Performance Testing of Component Based Software

In the context of this chapter, we take the perspective of the producer of a component-based system, who is interested in devising systematic ways to ascertain that a given distributed software architecture meets the performance requirements of their target users. Performance can be characterized in several different ways. **Latency** typically describes the delay between request and completion of an operation. **Throughput** denotes the number of operations that can be completed in a given period of time. **Scalability** identifies the dependency between the number of distributed system resources that can be used by a distributed application (typically number of hosts or processors) and latency or throughput. Despite the practical significance of these various aspects it is still not adequately understood how to test the performance of distributed applications.

As a consequence of the need for early evaluation of software performance and the weakness of testing, the majority of research efforts has focused on performance analysis models [23, 143, 140, 31, 39, 79] rather than testing techniques. This research shares in general the approach of translating architecture designs, mostly given in the Unified Modeling Language (UML [53]), to models suitable for analyzing performance, such as, Layered Queuing Networks (e.g. [140]), Stochastic Petri Nets (e.g. [31]) or stochastic process algebras (e.g. [143]). Estimates of performance are used to reveal flaws in the original architecture or to compare different architectures and architectural choices. Although models may give useful hints of the performance and help identify bottlenecks, they still tend to be rather inaccurate. Firstly, models generally ignore important details of the deployment environment. For example, performance differences may be significant when different databases or operating systems are used, but the complex characteristics of specific databases and operating systems are very seldom included in the models. Secondly, models often have to be tuned manually. For example, in the case of Layered Queued Networks, solving contention of CPU(s) requires, as input, the number of CPU cycles that each operation is expected to use. Tuning of this type of parameters is usually guessed by experience and as a result it is not easy to obtain precise models.

With the recent advances in distributed component technologies, such as J2EE [150] and CORBA [123], distributed systems are no longer built from scratch [77]. Modern distributed applications often integrate both off-the-shelf and legacy components, use services provided by third-parties, such as real-time market data provided by Bloomberg or Reuters, and rely on commercial databases to manage persistent data. Moreover, they are built on top of middleware products (hereafter referred to as *middleware*), i.e., middle-tier software that provides facilities and services to simplify distributed assembly of components, e.g., communication, synchronization, threading and load balancing facilities and transaction and security management services [78]. As a result of this trend, we have a class of distributed applications for which a considerable part of their implementation is already available when the architecture is defined, for example during the Elaboration phase of the Unified Process. In this chapter, we argue that this enables performance testing to be successfully applied at an early stage.

The main contribution of this chapter is the description and evaluation of a method for testing performance of distributed software in an early stage of development. The method is based on the observation that the middleware used to build a distributed application often determines the overall performance of the application. For example, middleware and databases usually contain the software for transaction and persistence management, remote communication primitives and threading policies, which have great impact on the different aspects of performance of distributed systems. However, we note that only the coupling between the middleware and the application architecture determines the actual performance. The same middleware may perform very differently in the context of different applications. Based on these observations, we propose using architecture designs to derive application-specific performance test cases that can be executed on the early available middleware platform a distributed application is built with. We argue that this allows empirical measurements of performance to be successfully done in the very early stages of the development process. Furthermore, we envision an interesting set of practical applications of this approach, that is: evaluation and selection of middleware for specific applications; evaluation and selection of off-the-shelf components; empirical evaluation and comparison of possible architectural choices; early configuration of applications; evaluation of the impact of new components on the evolution of existing applications.

The chapter is further structured as follows. Section 9.1 gives details of our approach to performance testing. Section 9.2 reports about the results of an empirical evaluation of the main hypothesis of our research, i.e., that the performance of distributed application can be successfully measured based on the early-available components. Section 9.3 discusses the limitations of our approach and sketches a possible integration with performance modeling techniques. Finally, Section 9.4 summarizes the contributions of the chapter and sketches our future research agenda.

9.1. Approach

In this section, we introduce our approach to early performance testing of distributed component-based software architectures. We also focus on the aspects of the problem that need further investigation. Our long-term goal is to provide an automated software environment that supports the application of the approach we describe below.

Our performance testing process consists of the following phases:

1. Selection of the use-case scenarios (hereafter referred to simply as *use-cases*) relevant to performance, given a set of architecture designs.
2. Mapping of the selected use-cases to the actual deployment technology and platform.
3. Generation of *stubs* of components that are not available in the early stages of the development life cycle, but are needed to implement the use cases.
4. Execution of the test, which in turn includes: deployment of the Application Under Test (AUT), creation of workload generators, initialization of the persistent data and reporting of performance measurements.

We now discuss the research problems and our approach to solving them for each of the above phases of the testing process.

9.1.1. Selecting Performance Use Cases

As it has been noticed by several researchers, such as Weyuker [174], the design of test suites for performance testing is radically different from the case of functional testing. In performance testing, the functional details of the test cases, i.e., the actual values of the inputs, are generally of limited importance. Table 9.1 classifies the main parameters relevant to performance testing of distributed applications. First, important concerns are traditionally associated with workloads and physical resources, e.g., the number of users, the frequencies of inputs, the duration of tests, the characteristics of the disks, the network bandwidth and the number and speed of CPU(s). Next, it is important to consider the middleware configuration, for which the table reports parameters in the case of J2EE-based middleware. Here, we do not comment further on workload, physical resource and middleware parameters, which are extensively discussed in the literature [174, 159, 111].

Other important parameters of performance testing in distributed settings are due to the interactions among distributed components and resources. Different ways of using facilities, services and resources of middleware and deployment environments are likely to yield different performance results. Performance will differ if the database is accessed many times or rarely. A given middleware may perform adequately for

Table 9.1.: Performance parameters

Category	Parameter
Workload	Number of clients Client request frequency Client request arrival rate Duration of the test
Physical resources	Number and speed of CPU(s) Speed of disks Network bandwidth
Middleware configuration	Thread pool size Database connection pool size Application component cache size JVM heap size Message queue buffer size Message queue persistence
Application specific	Interactions with the middleware - use of transaction management - use of the security service - component replication - component migration Interactions among components - remote method calls - asynchronous message deliveries Interactions with persistent data - database accesses

applications that stress persistence and quite badly for transactions. In some cases, a middleware may perform well or badly for different usage patterns of the same service. The last row of Table 9.1 classifies some of the relevant interactions in distributed settings according to whether they take place between the middleware and the components, among the components themselves[1] or to access persistent data in a database. In general, the performance of a particular application will be largely dependent on how the middleware primitives are being used to implement the application's functionality.

We argue that Application-specific test cases for performance should be given such

[1] Although interactions among distributed components map on interactions that take actually place at the middleware level, they are elicited at a different abstraction level and thus they are considered as a different category in our classification.

that the most relevant interactions specifically triggered by the AUT are covered. According to this principle, the generation of a meaningful test suite for performance testing can be based on either of two possible sources: previously recorded usage profiles or functional cases specified in the early development phases.

The former alternative is viable in cases of system upgrade. In this situation, "histories" of the actual usage profiles of the AUT are likely to be available because of the possibility that they have been recorded in the field. The synthesis of application specific workloads based on recorded usage profiles is a widely studied and fairly well understood research subject in the area of synthetic workload generation (e.g.[114, 157]).

When the development of a completely new application is the case, no recorded usage profile may exist. However, modern software processes tend to define the required functionality of an application under development in a set of scenarios and use cases. To build a meaningful performance test suite, we can associate a weight to each use case and generate a synthetic workload accordingly. The weight should express the importance of each use case in the specific test suite. Obviously to have a reliable evaluation of the performance characteristics of the application, we need to consider as many use cases as possible. This should be a minor problem because it is often the case that most of the use cases are available in early stages of a software process. For instance, the iterative and incremental development approaches (such as the Unified Software Development Process [54]) demand that the majority of use cases be available at the end of the early process iterations. In such settings, we can therefore assume that the software system developer can use these use cases to derive test cases to evaluate the performance of the final application, before starting with the implementation phase. On the base of the obtained results the developer can eventually revise the taken decisions in order to obtain better "expected" performance. To this end, several possibilities are available at this stage, (at a less expensive costs with respect to a late system refactoring, which might be required due to poor performance), such as, a revision of the architecture or a "re-"calibration of some choices concerning the middleware configuration.

9.1.2. Mapping Use Cases to Middleware

In the initial stages of the software process, software architectures are generally defined at a very abstract level. The early use-cases focus on describing the business logic, while they abstract the details of the deployment platform and technology. One of the strengths of our approach is indeed the possibility of driving software engineers through the intricate web of architectural choices, off-the-shelf components, distributed component technologies, middleware and deployment options, keeping the focus on the performance of the final product. The empirical measurements of performance may provide the base for comparing the possible alternatives. Consequently,

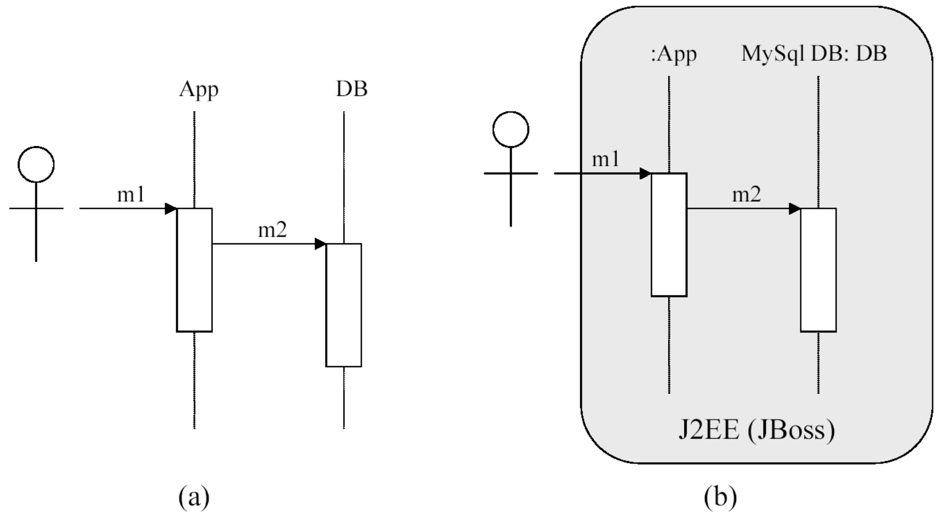

Figure 9.1.: An sample use-case (a) and part of a corresponding performance test case (b)

to define a performance test case, the abstract use-cases must be augmented with the following information:

- The mapping between the early available components (if any) and the components represented in the abstract use-cases;
- The distributed component technology and the actual middleware with respect to which the performance test is to be performed;
- The characteristics of the deployment of the abstract use-cases on the actual middleware platform, i.e., the specification of how the described component interactions take place through the selected component technology and middleware.

The two former requirements can be trivially addressed. For example, Fig. 9.1 (a) illustrates a sample abstract use-case, in which an actor accesses the service `m1` provided by the component `App`, which in turn uses the service `m2` provided by the component `DB`. Correspondingly, Fig. 9.1 (b) illustrates a performance test case in which: the component `DB` is instanced as the available MySql database engine, while the component `App` is not early available; the whole application is deployed using the J2EE component technology and the JBoss application server as middleware. The rest of this section discusses the problem of specifying the deployment characteristics.

At the architectural level, the properties of the component interactions can be described in terms of *software connectors*[2]. Recent studies (e.g., [118]) have investigated the role that software connectors may play in software design, showing that they may relevantly contribute to bridge the gap between the high-level application view of a software architecture and the implementation support provided by distributed component technologies and middleware. [121] attempts to classify software connectors and identifies a general set of *connector types*, their characteristics (*dimensions*) and the possible practical alternatives for each characteristic (*values*). For instance, the `procedure call` is identified as a connector type that enables communication and coordination among components; `synchronicity` is one of the dimensions of a procedure call connectors; and `synchronous` and `asynchronous` are the possible values of such dimension. When all dimensions of a connector type are assigned to specific values, the resulting instance of the connector type identifies a connector *specie*, e.g., the `remote method invocation` can be considered as a specie of the procedure call connector type. Our approach to the specification of the deployment characteristics leverages and extends the connector taxonomy of [121].

Up to now, we identified an initial set of connector types that specifically apply to the case of component interactions that take place through a J2EE compliant middleware. Giving values to the dimensions of these connectors allows for specifying the characteristics of the deployment of an abstract use-case on an actual middleware platform based on the J2EE specification. Specifically, we identified the following connector types: J2EE remote service, J2EE distributor, J2EE arbitrator and J2EE data access.

The *J2EE remote service* connector extends and specializes the procedure call connector type of [121]. This connector specifies the properties of the messages that flow among interacting components. We identified the following relevant dimensions for this connector:

- Synchronicity: A remote service can be either synchronous or asynchronous. Specifying a value for the synchronicity dimension allows to select if the service must be instanced as a synchronous method invocation or as an asynchronous event propagation, respectively.
- Parameters: This dimension specifies the number of parameters and their expected size in bytes. This allows for simulating the dependences between performance and the transfer of given amounts of data among components. Moreover, if the component that provides the service is one of the early available components, also types and values of the parameters must be provided to perform the actual invocation during the test. In this latter case, if the service is expected

[2]This is, for example, the spirit of the definition of software connectors given by Shaw and Garlan [91]: *connectors mediate interactions among components; that is, they establish the rules that govern component interaction and specify any auxiliary mechanisms required.*

to be invoked a number of times during the test, we can embed in the connector a strategy for choosing the values of the parameters:

1. a single value may be given. This value will be used every time the service is invoked during the test;
2. a list of values may be given. Each time the service is invoked a value of the list is sequentially selected;
3. a list of values and an associated probability distribution may be given. Each time the service is invoked a value of the list is selected sampling the distribution.

The *J2EE distributor* connector extends and specializes the distributor connector type of [121]. This connector allows to specify the deployment topology. We identified the following relevant dimensions for this connector:

- Connections: This dimension specifies the properties of the connections among the interacting components, i.e., the physical hosts on which they are to be deployed in the testing environment and the symbolic names used to retrieve the component factories through the naming service.
- Types. This dimension specifies the (expected) implementation type of the interacting components. Possible values are: client application, session bean, entity bean[3] and database table.
- Retrieving. This dimension specifies how to use the component factories (for components and interactions this is applicable to) for retrieving references to components. In particular, either the default or finder method can be specified (non standard retrieving methods of component factories are called *finders* in the J2EE terminology).

The *J2EE arbitrator* connector extends and specializes the arbitrator connector type of [121]. This connector specifies the participation in transactions and the security attributes of the component interactions. We identified the following relevant dimensions for this connector:

- Transactions: This dimension specifies the participation in transactions of a component interaction. Possible values are: none, starts and participates: none,

[3]Session beans are J2EE components that provide business services. Thus, session beans are often used as the interface between J2EE applications and client applications. Entity beans are J2EE components that represent persistent data within an application. Each database table is generally associated to an entity bean. The data in the entity bean are taken synchronized with the database. Thus, entity bean are often used as the interface between J2EE applications and databases.

if the interaction does not participate in any transaction; starts, if the interaction starts a new, possible nested, transaction; participates, if the interaction participates in the transaction of the caller.

- Security: This dimension specifies the security attributes of a component interaction. In particular, it specifies if services can be accessed by all users, specific users, or specific user groups, and which component is responsible for authentication in such two latter cases.

The *J2EE data access* connector extends and specializes the data access connector type of [121]. This connector mediates the communication between J2EE components and a database, specifying the structure of the database and how the interactions are handled. In particular, we identified the following relevant dimensions for this connector:

- Tables: This dimension specifies characteristics of the tables and their respective fields in the database.
- Relationships: This dimension specifies the presence of relationships among the tables in the database.
- Management: In J2EE components persistence can be handled either implementing the access functions (e.g., queries) in the component code (this is called bean managed persistence, BMP) or using standard mechanism embedded in the middleware (this is called container managed persistence, CMP).

Fig. 9.2 illustrates the application of connectors to the sample use-case of Fig. 9.1. As specified by the J2EE remote service connectors, the interactions `m1` and `m2` are both synchronous (i.e., they are assumed to be remote method invocations) and have just one input parameter. In the case of `m1`, only the parameter size is worth it, being the server component `App` not early available. Conversely, in the case of `m2`, also the actual value of the parameter is needed, being the database available. The specified parameter is the actual SQL code to be executed on the database and the "single value" strategy is used. The assumed database structure is specified in the J2EE data access connector `da2` and consists of a table (`T1`) with two integer fields (`F1` and `F2`) and no relationship, while the interactions between the component `App` and the MySql database are supposed to follow the bean managed persistence paradigm. The two J2EE distributor connectors, `d1` and `d2`, specify that the component `App` and the database are deployed on the same host (`host2`), while the client is on a different host (`host1`). The interface between the client and the component `App` is provided by a session bean EJB component and the interface between `App` and the database is handled by an entity bean EJB component. The retrieving strategy, when applicable, uses the standard methods provided by the platform. Finally, the J2EE arbitrator

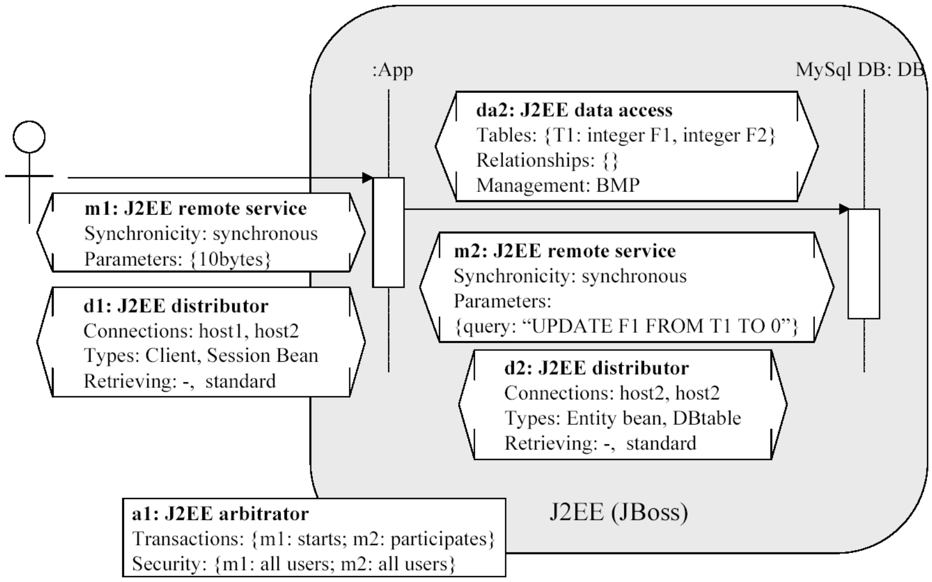

Figure 9.2.: A performance test case associated with the use-case in Fig. 9.1

connector specifies that m1 starts a transaction in which m2 participates and no special security policy is considered. The information given in Fig. 9.2 identifies a specific performance test case associated with the use-case in Fig. 9.1.

Notice that Fig. 9.2 is meant just for exemplification purpose and not to suggest an approach in which use-case diagrams must be annotated with connector information before testing. In a mature and engineered version of our approach, we envision the possibility that a tool analyses the abstract use-cases and extracts the simple list of alternatives for each interaction dimension. The performance engineer would then have the choice of selecting the best suited alternatives according to the performance requirements or test different alternatives to find out the one that works best (in a sort of what-if-analysis fashion). Software connectors provides the reasoning framework towards this goal. Furthermore, our current knowledge about all needed connector types and their dimensions is limited because it is based on a simple case in which we have experimented the application of the approach (Section 9.2 gives the details of this initial experience). We believe that we are on the right path, even though we are aware that further work is still needed to understand the many dimensions and species of software connectors and their relationships with the possible deployment technologies and platforms.

Putting MDA in the Picture. In Chapter 2 we extensively discussed the different alternatives for the modeling of a complex software system. Particularly interesting in the context of the work presented in this section will be possible development in the area of Model Driven Development (MDD). In fact, as the reader probably has already guessed, the approach described above can be considered a particular way of providing a development driven by the use and refinement of software models, defined using some kind of Architectural Description Language. As already discussed in Chapter 2 we think that the Software Architecture approach is the right approach to model complex system logically composed of many different interacting components. In particular the definition of the "connector" as a first class construct bring great logical improvement with respect to other modeling languages, permitting the definition of complex components interactions (see Chapter 2 for further details). The work that we described in this section it is a first attempt to identify connector features that should "easily" lead the developer to the deployment of the software architecture on the target platform through the phase of refinement that math an architectural connector to a real implementation depending on the selected platform. In Chapter 2 we discussed the relation between ADL and the upcoming version of the Unified Modeling Language (UML). The use of UML for the architectural description can bring great benefit to the approach that we described above, in particular in the context of the Model Driven Architecture methodology. Being able to express architectural concepts, such as that of connector, in UML will permit the reconsideration of the problem outlined above in term of PIM and PSM (see Chapter 2). In particular the transformation function between PIM and PSM (Figure 2.5) should solve the matching problem discussed. Being UML and MDA two greatly supported specifications, from the OMG consortium [13], we expect, in the near future, the availability of suitable tools that provide semi-automatic way (assisted by the developer) to make the mapping discussed in this section.

9.1.3. Generating Stubs

So far, we have suggested that early test cases of performance can be derived from use-cases and that software connectors can be exploited as a means to establish the correspondence between the abstract views provided by the use-cases and the concrete instances. However, to actually implement the test cases, we must also solve the problem that not all the application components that participate in the use-cases are available in the early stages of the development life cycle. For example, the components that implement the business logic are seldom available, although they participate in most of the use-cases. Our approach uses *stubs* in place of the missing components.

Stubs are fake versions of components that can be used instead of the corresponding components for instantiating the abstract use-cases. In our approach, stubs are specifically adjusted to use-cases, i.e., different use-cases will require different stubs of the same component. Stubs will only take care that the distributed interactions happen as specified and the other components are coherently exercised. Our idea of the engineered approach is that the needed stubs are automatically generated based on the information contained in use-cases elaborations and software connectors. For example, referring once again to Fig. 9.2, if the component *App* is not available, its stub would be implemented such that it is just able to receive the invocations of the service `m1` and consequently invokes the service `m2`, through the actual middleware. The actual SQL code embedded in the remote service connector of `m2` would be hard-coded in the stub. As for `m1`, it would contain empty code for the methods, but set the corresponding transaction behavior as specified. Of course, many functional details of *App* are generally not known and cannot be implemented in the stub. Normally, this will result in discrepancies between execution times within the stubs and the actual components that they simulate.

The main hypothesis of our work is that performance measurements in the presence of the stubs are good enough approximations of the actual performance of the final application. This descends from the observation that the available components, e.g., middleware and databases, embed the software that mainly impact performance. The coupling between such implementation support and the application-specific behavior can be extracted from the use-cases, while the implementation details of the business components remain negligible. In other words, we expect that the discrepancies of execution times within the stubs are orders of magnitude less than the impact of the interactions facilitated by middleware and persistence technology, such as databases. We report a first empirical assessment of this hypothesis in Section 9.2 of this chapter, but are aware that further empirical studies are needed.

The generation of the fake version can be made easier if we can use UML to describe the software architecture. The use of UML enables, in fact, the use of all the UML-based tools. A first interesting investigation in this direction can be found in [120]. In this work the authors propose different techniques to introduce concepts as connectors and architectural styles as a first order concepts inside an "extended" fully conform UML.

Putting MDA in the Picture. As discussed in the previous section, concerning architectural mapping issues, also for the step described in this section the novelties introduced in the area of UML and MDA can bring useful support to the stub generation. In fact we can consider the "fake" application as a particular application that can be obtained in the first steps of the refinement steps. In other words the application using stub elements can be considered as a less refined version of the final application in which the information concerning the business logic are missing. Therefore, the availability of tools supporting the MDA methodology should provide, for our objectives, an easily way to produce the version based on stubs component. It is worth to note that the deployment details for this version of the application should not be redefined later, but in some case only augmented with business logic dependent details, since the obtained performance evaluation will be strongly dependent to the specified deployment details such as localization of the components.

9.1.4. Executing the Test

Building the support to test execution shall mostly involve technical rather than scientific problems, at least once the research questions stated above have been answered. Part of the work consists of engineering the activities of mapping the use cases to deployment technologies and platforms, and generating the stubs to replace missing components. Also, we must automate deployment and implementation of workload generators, initialization of persistent data, execution of measurements and reporting of results.

In particular workload generator can be characterized in several different way, and many different workload can be found in literature (e.g.[66, 153]). It is a developer duty to choose the one that better represent the load that it expects for the application during the normal usage. Then after that the type of workload have been chosen, for instance from a list of possible different choice, and that the probability distributions have been associated to the relevant elements in the particular workload, it is possible to automatically generate the corresponding "application client" that generate invocations according to the chosen workload type and distributions.

9.2. Preliminary Assessment

This section empirically evaluates the core hypothesis of our research, i.e., that the performance of a distributed application can be successfully tested based on the middleware and/or off-the-shelf components that are available in the early stages of the software process. To this end, we conducted an experiment in a controlled environment. First, we considered a sample distributed application for which we had the whole implementation available. Then, we selected an abstract use-case of the application and implemented it as a test case based on the approach described in Section 9.1. Finally, we executed the performance test (with different amounts of application clients) on the early available components and compared the results with the performance measured on the actual application.

9.2.1. Experiment Setting

As for the target application, we considered the *Duke's Bank application* presented in the J2EE tutorial [51]. This application is distributed by Sun under a public license, thus we were able to obtain the full implementation easily. The Duke's bank

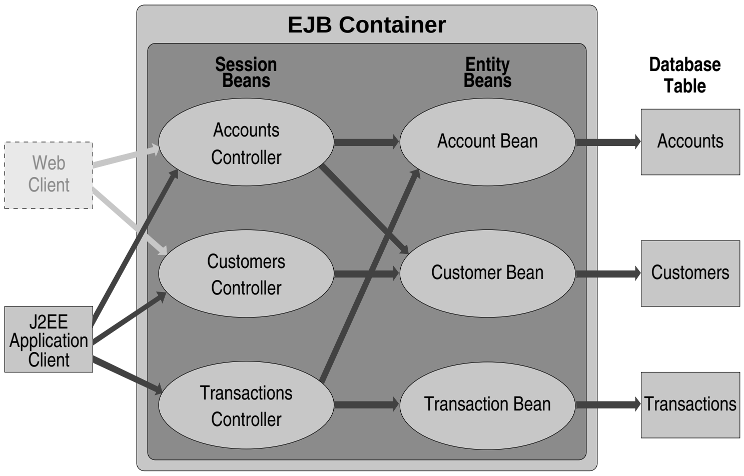

Figure 9.3.: The Duke's Bank application

application consists of 6,000 lines of Java code that is meant to exemplify all the main features of the J2EE platform, including the use of transactions and security. We consider the Duke's bank application to be adequately representative of medium-size component-based distributed applications. The Duke's bank application is referred to as DBApp in the remainder of this chapter.

The organization of the DBApp is given in Fig. 9.3 (borrowed from [51]). The application can be accessed by both Web and application clients. It consists of six EJB (Enterprise Java Beans [150]) components that handle operations issued by the users of a hypothetic bank. The six components can be associated with classes of operations that are related to bank accounts, customers and transactions, respectively. For each of these classes of operations a pair of session bean and entity bean is provided. Session beans are responsible for the interface towards the users and the entity beans handle the mapping of stateful components to underlying database table. The arrows represent the possible interaction patterns among the components. The EJBs that constitute the business components are deployed in a single container within the application server (which is part of the middleware). For the experiment we used the JBoss application server and the MySql database engine, running on the same machine.

Then, we selected a sample use-case that describes the transfer of funds between two bank accounts. Fig. 9.4 illustrates the selected use-case in UML. A client application uses the service `Transfer` provided by the DBApp. This service requires three input parameters, representing the two accounts and the amount of money, respectively involved in the transfer. The business components of the DBApp realize the

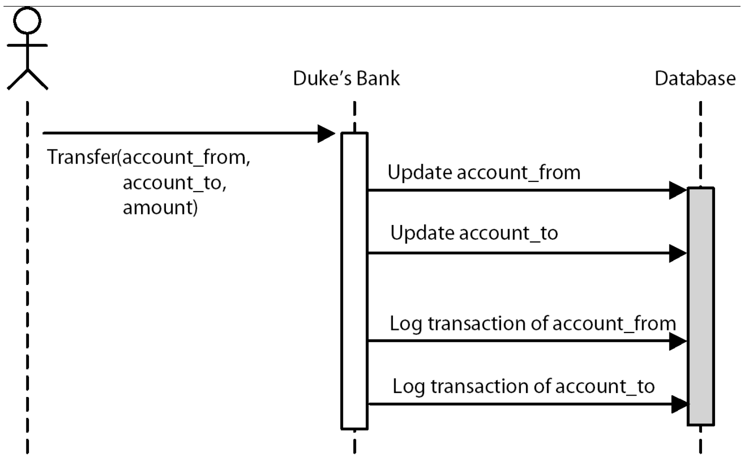

Figure 9.4.: A sample use-case for the Duke's Bank

service using the database for storing the persistent data: the database is invoked four times, for updating the balances of the two accounts and recording the details of the corresponding transactions. We assume that the database engine is software that is available early in the software process. Thus, for the test, we used the same database engine, table structure and SQL code than in the original application. This is why we represented the database as a shadowed box in the figure. Differently from the database, the business components of the DBApp are assumed to be not available, thus we had to generate corresponding stubs.

For implementing the stubs, we had to map the abstract use-case on the selected deployment technology, i.e., J2EE. We already commented on the role that software connectors may play in the mapping. As for the interaction between the clients and the DBApp, we specified that the service `Transfer` is invoked as a synchronous call and starts a new transaction. As for the interaction between the DBApp and the database, we specified that: the four invocations are synchronous calls that participate to the calling transaction and embed the actual SQL code; we set up the database factory such that the database connection is initialized for each call[4]; the DBApp uses entity beans and bean managed persistence to handle the interactions with the database tables. Based on this information, we implemented the stubs as needed to realize the interactions in the considered use-case and we deployed the test version of the DBApp (referred to as DBTest) on the JBoss application server.

Finally, we implemented a workload generator and initialized the persistent data

[4]Although this may sound as a bad implementation choice, we preferred to maintain the policy of the original application to avoid biases on the comparison.

in the database. The workload generator is able to activate a number of clients at the same time and takes care of measuring the average response time. For the persistent data, we instantiated the case in which each client withdraws money from its own account (i.e., there exists a bank account for each client) and deposits the corresponding amount to the account of a third party, which is supposed to be the same for all clients. This simulates the recurrent case in which a group of people is paying the same authority over the Internet. Incidentally, we notice that, in an automated test environment, initialization of persistent data would only require to specify the performance sensible part of the information, while the actual values in the database tables are generally of little importance. For example, in our case, only the number of elements in each table and the relationships with the instanced use-case, i.e., whether all clients access the same or a different table row, are the real concerns.

With reference to the performance parameters of Table 9.1, we generated a workload, to test both DBApp and DBTest, with increasing numbers of clients starting from one to one hundred. The two applications were deployed on a JBoss 3.0 application server running on a PC equipped with a Pentium III CPU at 1 GHz, 512 MB of RAM memory and the Linux operating system. To generate the workload we run the clients on a Sun Fire 880 equipped with 4 Sparc CPUs at 850 MHz and 8 GB of RAM. These two machines were connected via a private local area network with a bandwidth of 100 MBit/sec. For the stubs we used the same geographical distances as the components of the actual application. Moreover, in order to avoid influences among the experiments that could be caused by the contemporary existence of a lot of active session beans, we restarted the application server between two successive experiments. JBoss has been used running the default configuration. Finally, the specific setting concerning the particular use case, as already discussed in the previous paragraphs, foresaw the use of remote method calls between the components and the use of the transaction management service, in order to handle the data shared by the various beans consistently.

9.2.2. Experiment Results

We have executed both DBApp and DBTest for increasing numbers of clients and measured the latency for the test case. We repeated each single experiment 15 times and measured the average latency time. Fig. 9.5 shows the results of the experiments. It plots the latency time of both DBApp and DBTest against the number of clients, for all the repetitions of the experiment. We can see that the two curves are very near to each other. The average difference accounts for the 9.3% of the response time. The experiments also showed a low value for the standard deviation. The ratio between σ and the expectation results, in fact, definitively lower of the 0.15, both for the DBApp and for the DBTest.

The results of this experiment suggest the viability of our research because they

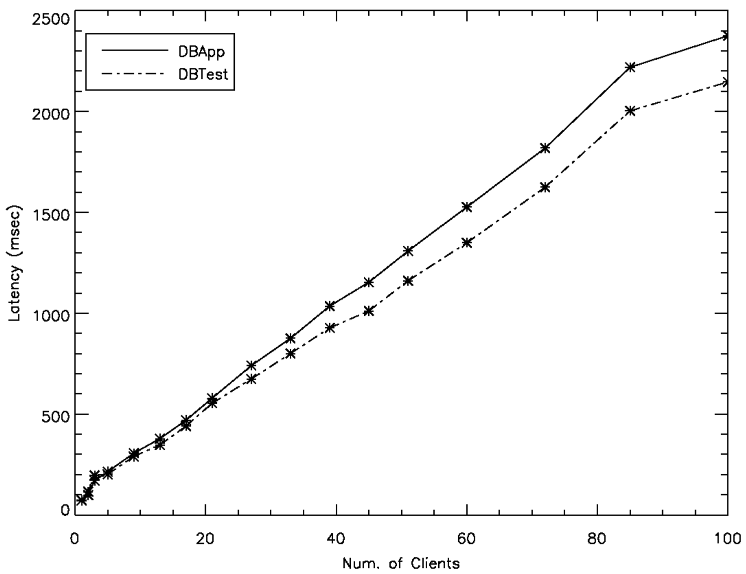

Figure 9.5.: Latency of DBApp and DBTest for increasing numbers of clients

witness that the performance of the DBApp in a specific use-case is well approximated by the DBTest, which is made out of the early-available components. However, although the first results are encouraging, we are aware that a single experiment cannot be generalized. We are now working on other experiments to cover the large set of alternatives of component-based distributed applications. We plan to experiment with different use-cases, sets of use-cases for the same test case, different management schemas for transactions and performance, different communication mechanisms such as asynchronous call, J2EE-based application server other than JBoss, CORBA-based middleware, other commercial databases and in the presence of other early-available components.

9.3. Scope and Extensions

Our results support the possibility that using stubs for the application code, but the real middleware and database proposed for the application, can provide useful information on the performance of a distributed application. This is particularly true for enterprise information system applications that are based on distributed component technologies, such as J2EE and CORBA. We have already commented that for this class of distributed applications the middleware is generally responsible for most of the implementation support relevant to performance, e.g., mechanisms for handling distributed communication, synchronization, persistence of data, transactions, load balancing and threading policies. Thus in most cases critical contention of resources and bottlenecks happen at the middleware level, while the execution time of the business components is negligible.

Our approach allows providers of this class of distributed applications to test whether, and to which extent, a given middleware may satisfy the performance requirements of an application that is under development. In this respect, our approach may perform better than pure benchmarking of middleware (e.g., [92, 108, 111]), because it enables application-specific evaluation, i.e., it generates test cases that take into account the specific needs of a particular business logic and application architectures. Moreover, the approach has a wider scope than solely testing the middleware. It can be generalized to test all components that are available at the beginning of the development process, for example, components acquired off-the-shelf by third parties. Based on the empirical measurements of performance, tuning of architectures and architectural choices may also be performed.

Despite these valuable benefits, however, we note that our approach cannot identify performance problems that are due to the specific implementation of late-available components. For example, if the final application is going to have a bottleneck in a business component that is under development, our approach has no chance to discover the bottleneck that would not be exhibited by a stub of the component. Performance analysis models remain the primary reference to pursue evaluation of performance in such cases.

Currently, we are studying the possibility of combining empirical testing and performance modeling, aiming at increasing the relative strengths of each approach. In the rest of this section we sketch the basic idea of this integration.

One of the problem of applying performance analysis to middleware-based distributed systems is that the middleware is in general very difficult to represent in the analysis models. For instance, let us consider the case in which one wants to provide a detailed performance analysis of the DBApp, i.e., the sample application used in Section 9.2. To this end, we ought to model the interactions among the business components of DBApp as well as the components and processes of the middleware that interact with DBApp. The latter include (and are not limited to) component proxies that marshal and unmarshal parameters of remote method invocations, the transaction manager that coordinates distributed transactions, the a database connectivity driver that facilitates interactions with the database, and the processes for automatic activation and deactivation of objects or components. Thus, although the application has a simple structure, the derivation of the correspondent analysis model becomes very costly.

We believe that this class of issues can be addressed by combining empirical testing and performance modeling. Currently we are investigating the possibility of using our approach as a preliminary steps to performance evaluation of systems features. In fact using the approach we can assess how the the application interacts with the specific middleware. In that manner we can derive useful values, expressing the performance characteristic of the specific application when deployed on the specific middleware, that can be successively used inside an analytical model to represent the middleware solving the problem of representing the it with rough models that generally lead to imprecise results.

9.4. Conclusions and Future Work

Distributed component technologies enforce the use of middleware, commercial databases and other off-the-shelf components and services. The software that implements these is available in the initial stages of a software process and moreover it generally embeds the software structures, mechanisms and services that mostly impact the performance in distributed settings. This chapter proposed to exploit the early availability of such software to accomplish empirical measurement of performance of distributed applications at architecture-definition-time. To the best of our knowledge, the approach proposed in this chapter is novel in software performance engineering.

This chapter fulfilled several goals. It described a novel approach to performance testing that is based on selecting performance relevant use-cases from the architecture designs, and instantiating and executing them as test cases on the early available software. It indicated important research directions towards engineering such approach, i.e.: the classification of performance-relevant distributed interactions as a base to select architecture use-cases; the investigation of software connectors as a mean to instantiate abstract use-cases on actual deployment technologies and platforms. It reported on experiments that show as the actual performance of a sample distributed application is well approximated by measurements based only on its early available components, thus supporting the main hypothesis of our research. It finally identified the scope of our approach and proposed a possible integration with performance modeling techniques aimed at relaxing its limitations.

Software performance testing of distributed applications has not been thoroughly investigated so far. The reason for this is, we believe, that testing techniques have traditionally been applied at the end of the software process. Conversely, the most critical performance faults are often injected very early, because of wrong architectural choices. Our research tackles this problem suggesting a method and a class of applications such that software performance can be tested in the very early stages of development. In the long term and as far as the early evaluation of middleware is concerned, we believe that empirical testing may outperform performance estimation models, being the former more precise and easier to use. Moreover, we envision the application of our ideas to a set of interesting practical cases:

- **Middleware selection:** The possibility of evaluating and selecting the best middleware for the performance of a specific application is reckoned important by many authors, as we already pointed out in Chapter 6. To this end, our approach provides a valuable support. Based on the abstract architecture designs, it allows to measure and compare the performance of a specific application for different middleware and middleware technologies.

- **COTS selection:** A central assumption of traditional testing techniques is that testers have complete knowledge of the software under test as well as of its requirements and execution environment. This is not the case for components off-the-shelf (COTS) that are produced independently and then deployed in environments not known in advance. Producers may fail in identifying all possible usage profiles of a component and therefore testing of the component in isolation (performed by producers) is generally not enough [146]. Limited to the performance concerns, our approach allows to test off-the-shelf components in the context of a specific application that is being developed. Thus, it can be used to complement the testing done by COTS providers and thus assist in selecting among several off-the-shelf components.

- **Iterative development:** Modern software processes prescribe iterative and incremental development in order to control risks linked to architectural choices (see e.g., the Unified Process [54]). Applications are incrementally developed in a number of iterations. During an iteration, a subset of the user requirements is fully implemented. This results in a working slice of the application that can be presently evaluated and, in the next iteration, extended to cover another part of the missing functionality. At the beginning of each iteration, new architectural decisions are generally made whose impact must be evaluated with respect to the current application slice. For performance concerns, our approach can be used when the life cycle architecture is established during the elaboration phase, because it allows to test the expected performance of a new software architecture based on the software that is initially available.

We are now performing further experiments to augment the empirical evidence of the viability of our approach and providing a wider coverage of the possible alternatives of component-based distributed applications. We are also working for engineering the approach, starting from the study of the research problems outlined in this chapter, and to this end we are considering the use of MDA based tools.

Part IV.

Conclusions

10. Conclusions and Future Work

This thesis discussed different methodologies for the functional and non-functional evaluation of systems implemented through the assembling of components, possibly externally acquired. I highlighted how the emergence of this new paradigm raises new problems, in particular with reference to the evaluation step. Assembling components to implement complex systems generally precludes, in fact, the possibility of using traditional analysis and testing techniques, since components are generally acquired from external providers and come without any information about internal characteristics. Moreover I discussed the basic importance of finding methodologies that can provide partial evaluation for a system under construction starting from the first steps of the development, since a belated identification of a flaw in the project generally implies the loss of huge amount of money, and in some cases leads to the failure of the project. Following this demand I discussed how some emerging modeling languages can be fruitfully used to make inferences on the final system behavior from a functional and non-functional point of view so reducing the risk for late detection of project flaws. In particular in Chapter 2 three main modeling paradigms have been introduced, such as Software Architecture, UML and MDA. I also highlighted both the respective relations of the three paradigms in the modeling of component based software systems, and why the concepts firstly introduced by Software Architecture, such as the concept of connector, are of basic importance in CB software development.

In this thesis two different chapters are dedicated to discuss two different proposals that try to make profit from an architectural description of a system. In particular in Chapter 7 I introduced the Component Deployment Testing Framework (CDT). The intention of the framework is to provide a simple means for the early codification of test cases defined to assess the conformance of components retrieved from the market, against those defined in the system architecture. By using the framework the system assembler can codify the test cases one time and successively use them to evaluate different components that could possibly be integrated in the system. The framework is based on the separation of the codification step from the adaptation step, reducing in that manner the probability of making errors. At the same time the defined test cases will be easily reusable when the system manager wants to substitute one component with a new one providing the same functionalities. Differently from other frameworks for component testing, CDT does not requires that the component developer implements, in the component, specific interfaces for testing purpose. Finally, CDT has been conceived to be used by the system assembler that, as deeply discussed in Chapter 7, must develop test cases on the base of his/her specifications with the advantage of having a component evaluation that reflects more closely the real final component usage and can also take into consideration the real environment.

The framework has been implemented in a proof of concept version and applied to the verification of a simple system (a dummy version of an FTP client) composed of two components, and it has given promising results. However the experiments highlighted that the adaptation phase, via the drawing up of the XMLAdapter file, could be quite difficult if there is a great difference between the component defined in the architecture and that selected to be possibly integrated in the system. This fact, suggested that the adaptation step should take advantage from the development of a graphical interface for the generation of the XMLAdapter. In the current version the framework is easy to use to functionally test a component in isolation, using stubs for the services required, or when it is integrated with other components providing the necessary services to the component itself. However the interactions of the components "behind the scene" are not checked during the execution of a test case. To improve the capability of the framework a sort of test-bench could be developed to track also the interactions among the components behind the component under test, so improving much more the capability of finding faults.

A second approach that strongly uses information from the architectural description of a system has been presented in Chapter 9. In this case the discussion is not focused on functional behavior but on performance evaluation. The starting point of the approach is the observation/hypothesis that the performance of a complex system implemented on top of a complex middleware is mainly consequence of the specific implementation and setting of the used middleware. For this reason we suggest that the use of analytical models generally leads to rough evaluations. Since middleware is a piece of code that is generally present when the system assembler starts to develop the architecture for the system, we proposed a methodology for the derivation of a prototype from the early architectural description of the system. Our hypothesis is that the obtained prototype, that uses the middleware in a similar way to how it will be done by the final system, reasonably approximate the behavior of the final system, at least for what concerns performance issues. The evaluation of the prototype is carried on by executing, on the derived prototype, opportunely defined test cases. We suggest that the selection of test cases, should be driven by the definition of operation profiles. In that manner the evaluation will be strongly focused on the scenarios that, with greater probability, will be activated in practice.

The approach has been used to foresee the performance of a simple application, provided by Sun Microsystems, deployed on the reference implementation of the J2EE application server. We compared the performance evaluation obtained running a test suite against the prototype derived from our approach, and against the real implementation. The results show that there is a strong relation among the two evaluations, that confirms the feasibility and quality of the proposed approach to performance prediction. However, even though the approach has gives encouraging results, it is not a panacea. In particular it is not applicable to find bottlenecks of an application when they reside in the business logic. For this reason we think that the approach proposed can be fruitfully used in combination with analytical models, and probably in that combination it can provide the best contribute. In fact, from our study we can conclude that the weakness of an approach are the point of strength of the other and vice versa. The investigation on how to better integrate the two different approaches is ongoing work.

Finally in Chapter 8 I discussed an approach for the behavioral evaluation of a system implemented assembling components. We start from the consideration that it is generally difficult to apply model-based testing to component based systems, as a consequence of the general black-box nature of external acquired components. So we propose a novel approach that tries to infer a model from the execution of the assembled system. The very general idea is to derive a set of test cases from an operational profile of the system under development. After that the system has been completely instantiated we execute the test cases on it, and from opportunely placed probes, we derive execution traces. From these traces and applying opportunely chosen synthesis algorithms we intend to derive a behavioral model that we can evaluate, using model checking for instance, to verify interesting system properties. The setting up of the approach to be really applicable still requires deeper investigations in different directions. Many difficulties both technical and theoretical must be solved. For instance, how to derive meaningful executions traces when concurrent processes are considered. At the same time it is necessary to derive from the executions a partial order among the observed traces to fruitfully apply the synthesis algorithms. Nevertheless I think that the approach can be an interesting contribute to the problem of obtaining a deep evaluation of a component-based software system. It is our intention to continue the investigation on this topic.

Bibliography

[1] The C2 style - UCI software architecture research. On line at: http://www.isr.uci.edu/architecture/c2.html.

[2] COM: Component Object Model Technologies. Available on-line at: http://www.microsoft.com/com/default.mspx.

[3] CORBA Component Model specifications. Available on-line at: http://www.omg.org/technology/documents/formal/components.htm.

[4] Darwin, an architectural description language. On line at: http://www-dse.doc.ic.ac.uk/Software/Darwin/.

[5] Distributed Objects & Components: CORBA ORBs. Available on-line at: http://www.cetus-links.org/oo_object_request_brokers.html.

[6] Enterprise Java Bean Technology. Available on-line at: http://java.sun.com/products/ejb/.

[7] iContract - the java dasign by contract tool. Available on-line at: http://www.reliable-systems.com/tools/iContract/iContract.htm.

[8] Intel philanthropic peer-to-peer program. On line at: www.intel.com/cure/.

[9] Java Remote Method Invocation (JavaRMI). Available on-line at: http://java.sun.com/products/jdk/rmi/.

[10] JUnit - testing resource for extreme programming. Available on-line at: http://www.junit.org.

[11] An MDA Tools list. http://www.modelbased.net/mda_tools.html.

[12] .Net resources. Available on-line at: http://www.microsoft.com/net/.

[13] Object Management Group (OMG). On line at: http://www.omg.org.

[14] The stanford rapide project. On line at: http://pavg.stanford.edu/rapide/.

[15] The Common Object Request Broker Architecture (CORBA) specifications. Available on-line at: http://www.corba.org/.

[16] The tracing vm. On line at: http://research.sun.com/people/mario/tracing-jvm/.

[17] Ubet: a requirements toolset for specifying requirements on the behavior of distributed systems. On line at: http://cm.bell-labs.com/cm/cs/what/ubet/.

[18] The Unified Modeling Language (UML). On line at: http://www.uml.org.

[19] *IEEE Computer*, volume 32. July 1999.

[20] Gregory Abowd, Robert Allen, and David Garlan. Using style to give meaning to software architecture. In *Proceedings of SIGSOFT '93: Foundations of Software Engineering*, volume 3 of *Software Engineering Notes*, pages 9–20. ACM Press, December 1993.

[21] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Transaction on Software Engineering and Methodology*, 6(3):213–249, July 1997.

[22] Colin Atkinson and Hans-Gerhard Groß. Built-in contract testing in model-driven, component-based development. In *Proceedings of Workshop on Component-Based Development Processes*, April 2002.

[23] Simonetta Balsamo, Paola Inverardi, and Calogero Mangano. An approach to performance evaluation of software architectures. In *Proceedings of the First International Workshop on Software and Performance (WOSP 1998)*, pages 178–190, 1998.

[24] Franck Barbier. *Testing Commercial-off-the-Shelf Components and Systems*, chapter COTS Component Testing through Built-In Test. Springer Verlag, to be published as hardcover book (December 2004).

[25] Luciano Baresi and Michal Young. Test oracles. Technical Report CIS-TR-01-02, University of Oregon, Dept. of Computer and Information Science, Eugene, Oregon, U.S.A., August 2001. On line at: http://www.cs.uoregon.edu/ michal/pubs/oracles.html.

[26] Francesca Basanieri, Antonia Bertolino, and Eda Marchetti. The Cow_Suite approach to planning and deriving test suites in uml projects. In *Proceedings of the International Conference on the Unified Modeling Language («UML»2002)*, LNCS 2460, pages 383–397, September 30 - October 4 2002. Dresden, Germany.

[27] Victor R. Basili and Richard W. Selby. Comparing the effectiveness of software testing strategies. *IEEE Transactions on Software Engineering*, 13(12):1278–1296, 1987.

[28] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley Professional, 1998.

[29] Ken Beck. *Test-Driven Development by Example*. Addison-Wesley, 2002.

[30] Boris Beizer. *Software Testing Techniques*. Van Nostrand, 2nd edition, 1990.

[31] Simona Bernardi, Susanna Donatelli, and José Merseguer. From UML sequence diagrams and statecharts to analysable Petri Nets models. In *Proceedings of the 3rd International Workshop on Software and Performance (WOSP 2002)*, pages 35–45, 2002.

[32] Philip A. Bernstein. Transaction processing monitors. *Communications of the ACM*, 33(11):75–86, November 1990.

[33] Philip A. Bernstein. Middleware: A model for distributed system services. *Communications of the ACM*, 39(2):86–98, February 1996.

[34] Antonia Bertolino. *SWEBOK: the Guide to the Software Engineering Body of Knowledge*, chapter Knowledge Area Description of Software Testing. Joint IEEE-ACM Software Engineering Coordination Committee, 2001. On line at: http://www.swebok.org.

[35] Antonia Bertolino. Software testing research and practice. In *Proceedings of the 10th Workshop on Abstract State Machine (ASM 2003)*, LNCS 2589, pages 1–21, March 3-7 2003. Taormina, Catania, Italy.

[36] Antonia Bertolino, Paola Inverardi, and Henry Muccini. An explorative journey from architectural tests definition downto code tets execution. In *Proceeding of 23rd International Conference on Software Engineering (ICSE 2001)*, pages 211–220, May 12-19 2001. Toronto, Canada.

[37] Antonia Bertolino, Paola Inverardi, and Henry Muccini. Formal methods in testing software architectures. In *Proceedings of the 3rd International School on Formal Methods (SFM 2003)*, LNCS 2804, pages 122–147, September 22-23 2003.

[38] Antonia Bertolino and Eda Marchetti. *Software Engineering*, volume 1, chapter A Brief Essay on Software Testing. IEEE Computer Society, to appear.

[39] Antonia Bertolino, Eda Marchetti, and Raffaela Mirandola. Real-time UML-based performance engineering to aid manager's decisions in multi-project planning. In *Proceedings of the 3rd International Workshop on Software and Performance (WOSP 2002)*, pages 251–261, New York, July 24–26 2002. ACM Press.

[40] Antonia Bertolino, Eda Marchetti, and Andrea Polini. Integrating "components" to test software components. In *Proceedings of 1st International Workshop on Testing and Analysis of Component Software at ETAPS 2003*, volume 82 of *ENTCS*, April 13th 2003. Warsaw - Poland.

[41] Antonia Bertolino and Raffaela Mirandola. Modeling and analysis of non-functional properties in component-based systems. In *Proceedings of 1st International Workshop on Testing and Analysis of Component Software at ETAPS 2003*, volume 82 of *ENTCS*, April 13th 2003. Warsaw - Poland.

[42] Antonia Bertolino and Raffaela Mirandola. CB-SPE tool: Putting component-based performance engineering into practice. In *Proceedings of the 7th International Symposium on Component-Based Software Engineering (CBSE 2004)*, volume 3054 of *LNCS*, pages 233–248, May 24-25 2004. Edinburgh, UK.

[43] Antonia Bertolino and Raffaela Mirandola. Software performance engineering of component-based systems. In *Proceeding of 4th International Workshop on Software Performance (WOSP 2004)*, volume 29 of *Software Engineering Notes*, pages 238–242. ACM Sigsoft, January 14-16 2004. Redwood Shores, California, USA.

[44] Antonia Bertolino and Andrea Polini. Re-thinking the development process of component-based software. In *Procediings ECBS 2002 Workshop On CBSE, Composing Systems From Components*, April 2002.

[45] Antonia Bertolino and Andrea Polini. WCT: a wrapper for component testing. In *Proceedings of International Workshop Fidji'2002*, volume 2604 of *LNCS*, pages 141–151, Luxembourg, November 28-29 2002.

[46] Antonia Bertolino and Andrea Polini. A framework for component deployment testing. In *Proceedings of 25th International Conference on Software Engineering (ICSE2003)*, pages 221–231, May 2003.

[47] Antonia Bertolino, Andrea Polini, Paola Inverardi, and Henry Muccini. Towards anti model-based testing. In *Supplemental Volume of the Proceedings of the International Conference on Dependable Systems and Networks (DSN 2004)*, pages 124–125, June 28th - July 1st 2004. Florence, Italy.

[48] Sami Beydeda and Volker Gruhn, editors. *Testing Commercial-off-the-Shelf Components and Systems*. Springer Verlag, to be published as hardcover book (December 2004).

[49] Robert V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley, 2000.

[50] Andrew D. Birrell and Bruce J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[51] Stephanie Bodoff et al. *The J2EE Tutorial*. Addison-Wesley, 2002.

[52] Barry Boehm, Ceci Albert, and Dan Port, editors. *3rd International Conference on COTS-Based Software Systems (ICCBSS 2004)*. February 1-4 2004. Redondo Beach, California - USA.

[53] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[54] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Software Development Process*. Addison-Wesley, 1999.

[55] David Braun, Jeff Sivils, Alex Shapiro, and Jerry Versteegh. What is UML. On line at: http://pigseye.kennesaw.edu/~dbraun/csis4650/A&D/UML_tutorial/, 2001.

[56] Lionel C. Briand, Yvan Labiche, and Yucong Miao. Towards the reverse engineering of uml sequence diagrams. In *Proceedings of 10th Working Conference on Reverse Engineering*, pages 57–66, November 13-17 2003. Victoria, B.C., Canada.

[57] Alan Brown. An introduction to Model Driven Architecture Part I: MDA and today's systems. *the Rational Edge*, January-February 2004. On line at: http://www-106.ibm.com/developersworks/rational/library/3100.html.

[58] Gary A. Bundell, Gareth Lee, John Morris, Kris Parker, and Peng Lam. A software component verification tool. In *Proceedings of International Conference on Software Methods and Tools (SMT2000)*, pages 137–146, Wollongong, Australia, November 6-10 2000.

[59] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, 1996.

[60] Xia Cai, Michael R. Lyu, and Kam-Fai Wong. *Testing Commercial-off-the-Shelf Components and Systems*, chapter A Generic Environment for COTS Testing and Quality Prediction. Springer Verlag, to be published as hardcover book (December 2004).

[61] Mauro Caporuscio, Antonio Carzaniga, and Alexander L. Wolf. Design and evaluation of a support service for mobile, wireless publish/subscribe applications. *IEEE Transactions on Software Engineering*, 29(12):1059–1071, December 2003.

[62] Alejandra Cechich and Macario Polo. *Testing Commercial-off-the-Shelf Components and Systems*, chapter COTS Components Testing through Aspect-based Metadata. Springer Verlag, to be published as hardcover book (December 2004).

[63] John Cheesman and John Daniels. *UML Components - a Simple Process for Specifying Component-Based Software*. Addison-Wesley, 2001.

[64] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, January 2000.

[65] Alistair Cockburn. *Writing Effective Use Cases*. Addison-Wesley, 2001.

[66] Vittorio Cortellessa and Raffaela Mirandola. PRIMA-UML: a performance validation incremental methodology on early UML diagrams. *Science of Computer Programming*, 44:101–129, 2002.

[67] Ivica Crnkovic. Component-based software engineering - new challenges in software development. *Software Focus*, December 2001. John Wiley & Sons Editors.

[68] Ivica Crnkovic and Magnus Larsson, editors. *Building Reliable Component-Based Software System*. Artech House Publisher, 2002.

[69] Ivica Crnkovic, Heinz Schmidt, Judith Stafford, and Kurt Wallnau, editors. *The Journal of Systems and Software - Special issue on CBSE*, volume 65. 2003.

[70] Ivica Crnkovic, Heinz Schmidt, Judith Stafford, and Kurt C. Wallnau, editors. *7th International Symposium on CBSE*. May 24-25 2004. Edimburgh, Scotland - UK.

[71] Ivica Crnkovic, Judith Stafford, and Stig Larsson, editors. *Proceedings of ECBS 2002 Workshop on CBSE, Composing System From Components*. April 10-11 2002. Lund, Sweden.

[72] Gianpaolo Cugola, Elisabetta Di Nitto, and Alfonso Fuggetta. The jedi event-based infrastructure and its application to the development of the opss wfms. *IEEE Transactions on Software Engineering*, 27(9):827–850, September 2001.

[73] Giovanni Denaro, Andrea Polini, and Wolfgang Emmerich. Early performance testing of distributed software applications. In *Proceedings of the 4th International Workshop on Software Performance (WOSP 2004)*, volume 29 of *Software Engineering Notes*, pages 94–103. ACM Sigsoft, January 14-16 2004. Redwood Shores, California.

[74] Giovanni Denaro, Andrea Polini, and Wolfgang Emmerich. *Testing Commercial-off-the-Shelf Components and Systems*, chapter Performance Testing of Distributed Component Architectures. Springer Verlag, to be published as hardcover book (December 2004).

[75] Esdger W. Dijkstra. Notes on structured programming. Technical Report T.H.-Report 70-WSK-03, Technological University of Eindhoven, April 1970. On line at: http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.pdf.

[76] Wolfgang Emmerich. *Engineering Distributed Objects.* John-Wiley & Sons, 2000.

[77] Wolfgang Emmerich. Software engineering and middleware: a roadmap. In *Proceedings of the 22th International Conference on Software Engineering (ICSE-00)*, pages 117–132. ACM Press, 2000.

[78] Wolfgang Emmerich. Distributed component technologies and their software engineering implications. In *Proceedings of the 24th International Conference on Software Engineering (ICSE-02)*, pages 537–546. ACM Press, 2002.

[79] Wolfgang Emmerich and James Skene. Model driven performance analysis of enterprise information systems. In *Proceedings of the International Workshop on Testing and Analysis of Component-Based Systems(TACoS'03)*, 2003.

[80] Wolfgang Emmerich and Alexander Wolf, editors. *2nd International Working Conference on Component Deployment (CD 2004).* May 20-21 2004. Edimburgh, Scotland - UK.

[81] Hans-Erik Eriksson, Magnus Penker, Brian Lyons, and David Fado. *UML 2 Toolkit.*

[82] Jacky Estublier and Jean-Marie Favre. *Building Reliable Component-Based Software System*, chapter Component Models and Technology, pages 57–86. Artech House Publisher, 2002.

[83] Colin Atkinson et al. *Component-based Product Line Engineering with UML.* Addison-Wesley, 2001.

[84] Greg Franks, Alex Hubbard, Shikharesh Majumdar, John Neilson, Dorina Petriu, Jerome Rolia, and Murray Woodside. A toolset for performance engineering and software design of client-server systems. *Performance Evaluation*, 24(1-2):117–136, November 1995.

[85] Michael A. Friedman and Jeffrey M. Voas. *Software Assessment: reliability, safety, testability*. John Wiley & sons, 1995.

[86] Erich Gamma, Richard Helm, Robert Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[87] Jerry Gao, Kamal Gupta, Shalini Gupta, and Simon Shim. On building testable software components. In J. Dean and A. Gravel, editors, *Proceedings of International Conference on Component-based Software System, 2002*, pages 108–121, 2002.

[88] David Garlan. Software architecture: a roadmap. In Anthony Finkelstein, editor, *Foundation of Software Engineering*, pages 91–101. ACM Press, 2000.

[89] David Garlan and Robert Allen. Formalizing architectural connection. In *Proceedings of the 16th International Conference on Software Engineering*, pages 71–80. IEEE Computer Society Press, May, 16-21 1994. Sorrento, Italy.

[90] David Garlan, Robert Allen, and John Ockerbloom. Architectural mismatch or why it's hard to build system out of existing parts. In *Proceedings 17th International Conference on Software Enginneering*, pages 179–185, April 1995.

[91] David Garlan and Mary Shaw. *Software Architecture: Perspective on an Emerging Discipline*. Prentice-Hall, 1996.

[92] Ian Gorton and Anna Liu. Software component quality assessment in practice: successes and practical impediments. In *Proceedings of the 24th International Conference on Software Engineering (ICSE 2002)*, pages 555–558, New York, 2002. ACM Press.

[93] Dorothy Graham. Testing object-oriented systems. In *Ovum Evaluates: Software Testing Tools*. Ovum Ltd., February 1996.

[94] Vincenzo Grassi and Raffaela Mirandola. Towards automatic compositional performance analysis of component-based systems. In *Proceeding of 4th International Workshop on Software Performance (WOSP 2004)*, volume 29 of *Software Engineering Notes*, pages 59–63. ACM Sigsoft, January 14-16 2004. Redwood Shores, California, USA.

[95] Hans-Gerhard Groß, Ina Schieferdecker, and George Din. *Testing Commercial-off-the-Shelf Components and Systems*, chapter Modeling and Implementation of Built-In Contract Test. Springer Verlag, to be published as hardcover book (December 2004).

[96] Dick Hamlet. Continuity in software systems. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2002)*, volume 27 of *Software Engineering Notes*, pages 196–200. ACM Sigsoft, July 22-24 2002. Roma - Italy.

[97] Dick Hamlet. Software component synthesis theory: A subdomain-testing approach. Seminar at Information Science and Technology Institute (ISTI/CNR), July 5th 2004. Pisa, Italy.

[98] Alan Hartman and Kenneth Nagin. The AGEDIS tools for model based testing. In *Proceedings of International Symposium on Software Testing and Analysis (ISSTA 2004)*, pages 129–132, July 11-14 2004. Boston, Massachusetts, USA.

[99] Rob Hierons. Testing from a Z specification. *Software Testing, Verification and Reliability*, 7:19–33, 1997.

[100] Charles A. H. Hoare. *Communicating Sequential Process*. Prentice Hall, 1985.

[101] Christine Hofmeister, Robert Nord, and Dilip Soni. *Applied Software Architecture*. Addison Wesley Professional, 1999.

[102] Paola Inverardi and Massimo Tivoli. *Formal Methods for the Design of Computer, Communication and Software Systems: Software Architecture*, chapter Software Architecture for Correct Components Assembly. Number 2804 in LNCS. Springer, 2003.

[103] Gregor Kiczales and Jim des Rivières. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[104] Anneke Kleepe, Jos Warmer, and Wim Bast. *MDA Explained - The Model Driven Architecture: Practice and Promise*. Addison-Wesley, 2003.

[105] Philippe B. Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(6):42–50, November 1995.

[106] Philippe B. Kruchten. *The Rational Unified Process: An Introduction*. Addison Wesley Longman, 2000.

[107] Scott M. Lewandowski. Frameworks for component-based client/server computing. *ACM Computing Surveys*, 30(1):3–27, March 1998.

[108] Cheng J. Lin, Alberto Avritzer, Elaine Weyuker, and Sai-Lai Lo. Issues in interoperability and performance verification in a multi-orb telecommunications environment. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN 2000)*, pages 567–575, 2000.

[109] Jacques-Louis Lions. Ariane 5, flight 501 failure, report by the inquiry board. Available at: http://java.sun.com/people/jag/Ariane5.html, 1996.

[110] Bev Littlewood, Peter Popov, Lorenzo Strigini, and Nick Shryane. Modelling the effects of combining diverse software fault detection techniques. *IEEE Transaction on Software Engineering*, 26(12):1157–1167, 2000.

[111] Yan Liu, Ian Gorton, Anna Liu, Ning Jiang, and Shiping Chen. Designing a test suite for empirically-based middleware performance prediction. In *Fortieth International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*, Sydney, Australia, 2002. ACS.

[112] David C. Luckham, John L. Kenney, Larry M. Augustin, James Vera, Doug Bryan, and Walter Mann. Specification and analysis of system architecture using rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, April 1995.

[113] Frank Lüders, Ivica Crnkovic, and Andreas Sjögren. A componet-based software architecture for industrial control. In *Proceedings of WICSA 2002*, pages 193–204, August, 25-30 2002. Montréal, Canada.

[114] Wey lun Kao and Ravishankar K. Iyer. A user-oriented synthetic workload generator. In *12th International Conference on Distributed Computing Systems (ICDCS '92)*, pages 270–277. IEEE Computer Society Press, June 1992.

[115] Jeff Magee, Naranker Dulay, Susan Eisenbach, and Jeff Kramer. Specifying distributed software architectures. In *Proceedings 5th European Software Engineering Conference (ESEC 95)*, pages 137–153, Sitges, Spain, 1995.

[116] Leonardo Mariani. Behavior capture and test for verifying evolving component-based systems. In *Proceeding of 26th International Conference on Software Engineering (ICSE 2004)*, pages 78–80, May 23-28 2004. Edinburgh, Scotland, UK.

[117] Doug McIllroy. Mass produced software components. In P. Naur and B. Randall, editors, *Software Engineering: Report on a Conference by the NATO Science Committee*, pages 138–155, 1969.

[118] Nenad Medvidovic, Eric M. Dashofy, and Richard N. Taylor. The role of middleware in architecture-based software development. *International Journal of Software Engineering and Knowledge Engineering*, 13(4):367–393, 2003.

[119] Nenad Medvidovic, Peyman Oreizy, Jason E. Robbins, and Richard N. Taylor. Using object-oriented typing to support architectural design in the C2 style. In *Proceedings of 4th Symposium on the Fundation of Software Engineering (FSE4)*, pages 24–32. ACM Press, October 1996. San Francisco, California (USA).

[120] Nenad Medvidovic, David S. Rosenblum, David F. Redmiles, and Jason E. Robbins. Modeling software architectures in the unified modeling language. *ACM Transcations on Software Engineering and Methodology*, 11(1):2–57, January 2002.

[121] Nikunj R. Mehta, Nenad Medvidovic, and Sandeep Phadke. Towards a taxonomy of software connectors. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000)*, pages 178–187. ACM Press, 2000.

[122] Atif M. Memon. *Testing Commercial-off-the-Shelf Components and Systems*, chapter Developer's Role in Making Testable Components. Springer Verlag, to be published as hardcover book (December 2004).

[123] Philippe Merle. CORBA 3.0 new components chapters. Technical report, TC Document ptc/2001-11-03, Object Management Group, 2001.

[124] Bertrand Meyer. *Eiffel - The Language*. Series in Computer Science. Prentice Hall, 1990.

[125] Bertrand Meyer. Applying design by contract. *IEEE Computer*, 25(10):40–51, October 1992.

[126] Bertrand Meyer, Christine Mingins, and Heinz Schmidt. Trusted components for the software industry.
Available at: http://trusted-components.org/documents/tc_original_paper.html.

[127] H. D. Mills. Top-Down Programming in Large Systems. In R. Ruskin, editor, *Debugging Techniques in Large Systems*. Prentice Hall, 1971.

[128] Robert Monroe, Drew Kompanek, Ralph Melton, and David Garlan. Architectural style, design patterns, and objects. *IEEE Software*, 14(1):43–52, January 1997.

[129] Richard Monson-Haefel. *Enterprise JavaBeans - Developing Enterprise Java Components*. O'Reilly, 3rd edition, 2001.

[130] John Morris, Gareth Lee, Kris Parker, Gary A. Bundell, and Chiou P. Lam. Software component certification. *IEEE Computer*, 34(9):30–36, September 2001.

[131] Henry Muccini. *Software Architecture for Testing, Coordination and Views Model Checking*. PhD thesis, Università degli Studi di Roma - La Sapienza, 2002.

[132] Henry Muccini, Antonia Bertolino, and Paola Inverardi. Using software architecture for code testing. *IEEE Transaction on Software Engineering*, 30(3):160–171, March 2004.

[133] John D. Musa. Operational profiles in software-reliability engineering. *IEEE Software*, 10(2):14 – 32, March 1993.

[134] Glenford J. Myers. *The Art of Software Testing*. John Wiley & sons, 1979.

[135] Elisabetta Di Nitto and David Rosemblum. Exploiting adls to specify architectural styles induced by middleware infrastructures. In *Proceedings of the 21st international conference on Software engineering (ICSE 1999)*, pages 13–22, May 16 - 22 1999. Los Angeles, California, United States.

[136] Alessandro Orso, Mary J. Harrold, and David Rosenblum. Component metadata for software engineering tasks. In Wolfgang Emmerich and Simon Tai, editors, *Proceeedings of International Conference on Engineering Distributed Objects 2000*, LNCS 1999, pages 129–144, 2000.

[137] Alessandro Orso, Mary Jean Harrold, David Rosenblum, Greg Rothermel, Mary Lou Soffa, and Hyunsook Doo. Using component metadata to support the regression testing of component-based software. In *Proceedings of the International Conference on Software Maintenance (ICSM2001)*, pages 716–725, Florence, Italy, November 6-10 2001.

[138] Rajesh G. Parekh and Vasant Honavar. *The Handbook of Natural Language Processing*, chapter Grammar Inference, Automata Induction, and Language Acquisition, pages 727–764. Marcel Dekker Inc., 2000.

[139] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.

[140] Dorina Petriu, Christiane Shousha, and Anant Jalnapurkar. Architecture-based performance analysis applied to a telecommunication system. *IEEE Transactions on Software Engineering*, 26(11):1049–1065, 2000.

[141] Shari Lawrence Pfleeger. *Software Engineering - theory and practice.* Prentice Hall Internetional, 2nd edition, 2001.

[142] Andrea Polini and Antonia Bertolino. *Testing Commercial-off-the-Shelf Components and Systems*, chapter A User-Oriented Framework for Component Deployment Testing. Springer Verlag, to be published as hardcover book (December 2004).

[143] Rob Pooley. Using UML to derive stochastic process algebra models. In *Proceedings of the 15th UK Performance Engineering Workshop (UKPEW)*, pages 23–34, 1999.

[144] Jerome A. Rolia and Kenneth C. Sevcik. The method of layers. *IEEE Transactions on Software Engineering*, 21(8):689–700, August 1995.

[145] David Rosenblum. Adequate testing of component-based software. Technical Report UCI-ICS-97-34, University of California at Irvine, 1997.

[146] David Rosenblum. Challenges in exploiting architectural models for software testing. In *Proceedings of the International Workshop on the Role of Software Architecture in Testing and Analysis (ROSATEA)*, 1998.

[147] Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture.* John Wiley and Sons, 2000.

[148] Sibylle Schupp, Marcin Zalewski, and Kyle Ross. Rapid performance prediction for library components. In *Proceeding of 4th International Workshop on Software Performance (WOSP 2004)*, volume 29 of *Software Engineering Notes*, pages 69–73. ACM Sigsoft, January 14-16 2004. Redwood Shores, California, USA.

[149] Bran Selic. The Pragmatics of Model-Driven Development. *IEEE Software*, 20(5):19–25, September 2003.

[150] Bill Shannon. Java 2 Platform Enterprise Edition specification, 1.4 - final release. Technical report, Sun Microsystems, Inc., November 24th 2003.

[151] Mary Shaw, Robert DeLine, Daniel Klein, Theodore Ross, David Young, and Gregory Zelesnik. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering*, 21(4):314–335, April 1995.

[152] Avi Silberschatz, Peter Galvin, and Greg Gagne. *Operating Systems Concepts.* Addison-Wesley, 6th edition, 2002.

[153] James Skene and Wolfgang Emmerich. A model-driven approach to non-functional analysis of software architectures. In *Proceedings of 18th IEEE International Conference on Automated Software Engineering (ASE 2003)*, pages 236–239, Montreal, Canada, October 6-10 2003.

[154] Carol Sliwa. Sidebar: Waiting for UML 2.0. On line at: http://www.computerworld.com/printthis/2004/0,4814,91325,00.html, March 22nd 2004.

[155] Connie U. Smith. *Performance Engineering of Software Systems.* Addison-Wesley, 1990.

[156] Dilip Soni, Robert L. Nord, and Christine Hofmeister. Software architecture in industrial applications. In *Proceedings of International Conference on Software Engineering*, pages 196–207, Seattle, Washington USA, May 1995. ACM Press.

[157] K. Sreenivasan and A. Kleinman. On the construction of a representative synthetic workload. *Communications of the ACM*, 17(3):127–133, March 1974.

[158] Judith A. Stafford and Alexander L. Wolf. Annotating components to support component-based static analyses of software systems. In *Proceedings of the Grace Hopper Celebration of Women in Computing*, 2001.

[159] B. Subraya and S. Subrahmanya. Object driven performance testing of Web applications. In *Proceedings of the First Asia-Pacific Conference on Quality Software (APAQS'00)*, 2000.

[160] Clemens Szyperski, Dominik Gruntz, and Stephen Murer. *Component Software - Beyond Object-Oriented Programming.* Addison-Wesley, 2nd edition, 2002.

[161] Massimo Tivoli and Marco Autili. SYNTHESIS: a tool for synthesizing "correct" and protocol-enhanced adaptors. Technical report, University of L'Aquila, Computer Science Department, October 2004. On line at: http://www.di.univaq.it/tivoli/LastSynthesis.pdf.

[162] Jan Tretmans. Conformance testing with labeled transition systems: Implementation, relations and test generation. *Computer Networks and ISDN Systems*, 29:49–79.

[163] Sebastián Uchitel, Jeff Kramer, and Jeff Magee. Detecting implied scenarios in message sequence chart specifications. In *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE 2001)*, pages 74–82, September 10 - 14 2001. Vienna, Austria.

[164] Sebastian Uchitel, Jeff Kramer, and Jeff Magee. Synthesis of behavorial models from scenarios. *IEEE Transaction on Software Engineering*, 29(2):99–115, February 2003.

[165] Amjad Umar. *Object-Oriented Client/Server Internet Environments*. Prentice Hall, 1997.

[166] Tom Verdickt, Bart Dhoedt, and Frank Gielen. Incorporating SPE into MDA: Including middleware performance detains into system models. In *Proceeding of 4th International Workshop on Software Performance (WOSP 2004)*, volume 29 of *Software Engineering Notes*, pages 120–124. ACM Sigsoft, January 14-16 2004. Redwood Shores, California, USA.

[167] Jeffrey Voas. Certifying off-the-shelf software components. *IEEE Computer*, 31(6):53–59, June 1998.

[168] Jeffrey Voas. Developing a usage-based software certification process. *IEEE Computer*, 33(8):32–37, August 2000.

[169] AA. VV. UML 2.0 Superstructure Specification. Technical report, Object Management Group, August 2003.

[170] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. A note on distributed computing. Technical Report TR-94-29, Sun Microsystems, November 1994. Available at: http://research.sun.com/techrep/1994/abstract-29.html.

[171] Yingxu Wang, Graham King, and Hakan Wickburg. A method for built-in tests in component-based software maintenance. In *Proceedings of the IEEE International Conference on Software Maintenance and Reengineering (ICSMR99)*, pages 186–189, 1999.

[172] Jos Warmer and Anneke Kleepe. *The Object Costraint Language: Getting your models ready fro MDA*. Addison-Wesley, 2nd edition, 2003.

[173] Elaine Weyuker. Testing component-based software: A cautionary tale. *IEEE Software*, 15(5):54–59, September/October 1998.

[174] Elaine Weyuker and Filippos Vokolos. Experience with performance testing of software systems: issues, an approach, and case study. *IEEE Transactions on Software Engineering*, 26(12):1147–1156, 2000.

[175] Elaine J. Weyuker. Translability and decidability questions for restricted classes of program schemas. *SIAM Journal on Computers*, 8(4):587–598, 1979.

[176] Elaine J. Weyuker. On testing non-testable program. *The Computer Journal*, 25(4):703–711, 1982.

[177] John Whaley, Michael C. Martin, and Monica S. Lam. Automatic extraction of object-oriented component interfaces. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2002)*, pages 218–228, July 22-24 2002. Roma, Italy.

[178] Xiuping Wu and Murray Woodside. Performance modeling from software components. In *Proceeding of 4th International Workshop on Software Performance (WOSP 2004)*, volume 29 of *Software Engineering Notes*, pages 290–301. ACM Sigsoft, January 14-16 2004. Redwood Shores, California, USA.