A Taxonomy of the Quality Attributes for Distributed Applications

Jorge Enrique Pérez-Martínez and Almudena Sierra-Alonso

University Rey Juan Carlos
E.S. of Experimental Sciences and Technology
C/ Tulipán s/n, 28933 Móstoles, Madrid, Spain

{j.perez, a.sierra}@escet.urjc.es

Abstract. The software engineering community has paid little attention to non-functional requirements, or quality attributes, compared with studies performed on capture, analysis and validation of functional requirements. This circumstance becomes more intense in the case of distributed applications. In these applications we have to take into account, besides the quality attributes such as correctness, robustness, extendibility, reusability, compatibility, efficiency, portability and ease of use, others like reliability, scalability, transparency, security, interoperability, concurrency, etc. In this work we will show how these last attributes are related to different abstractions that coexist in the problem domain. To achieve this goal, we have established a taxonomy of quality attributes of distributed applications and have determined the set of necessary services to support such attributes.

Keywords: quality attribute, distributed application, intrinsic attributes, specific attributes, mechanisms, services.

1. Introduction

Kotonya and Sommerville (1998) state that "non-functional requirements define the overall qualities or attributes of the resulting system". Until recently, the software engineering community has paid little attention to non-functional requirements compared with studies performed on capture, analysis and validation of functional requirements (Pressman, 2001; Sommerville, 2001). According to Bass, Clements and Kazman (1998, p. 76), we will designate those non-functional requirements with the term "quality attributes". However, nobody doubts that attributes like correctness, robustness, extendibility, reusability, compatibility, efficiency, portability and ease of use (Meyer, 1997) define the quality of the software, and such quality is an unavoidable objective of any software development. There is no place for compromise between quality and any other aspect. In the distributed applications domain, software development must also cover aspects like reliability, scalability, transparency, security, interoperability, concurrency, etc. (Emmerich, 2000).

In this work we propose a taxonomy of the quality attributes of distributed applications. This taxonomy will allow us to determine the necessary services to support each attribute.

The content of this work is organised as follows. Section 2 describes and models the concept of a distributed system. Section 3 presents the taxonomy that we have defined for the quality attributes of distributed applications. In section 4 we study the necessary services to support those attributes and we model them using a UML class diagram. Finally, section 5 summaries main conclusions and future work.

2. Modelling and describing a distributed system

In fig. 1 we represent, using a UML class diagram (Rumbaugh, Jacobson and Booch, 1999), what can be considered the fundamental abstractions of a distributed system. Basically, a distributed system is a set of concurrent processes that access a set of shared resources with a set of access characteristics like security, consistency, transparency, etc. These processes are executed in autonomous computers connected in a network, and are able to communicate with one another.

From this figure, we will study how different quality attributes relate to the represented abstractions and will determine the services needed to support them.

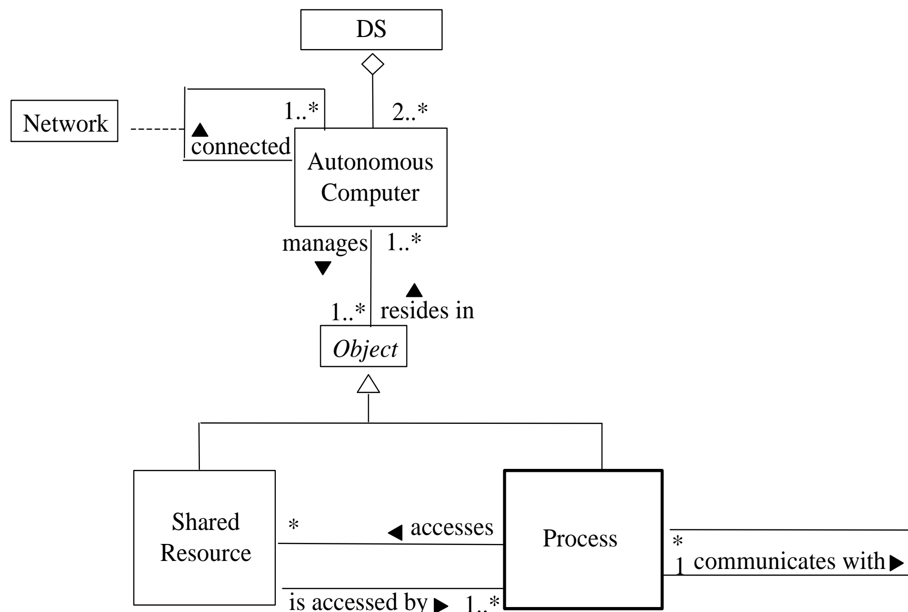

Fig. 1. Main abstractions in a distributed system.

3. A taxonomy of the quality attributes in the distributed applications

When trying to describe the non-functional requirements of a distributed system, different authors cite different sets of them. For example, Coulouris, Dollimore and Kindberg (1994) define the following as the key characteristics of any distributed system: resource sharing, openness, concurrency, scalability, fault-tolerance and transparency. Emmerich (2000) indicates that the non-functional requirements leading us to adopt a distributed architecture are: resource sharing, scalability, fault-tolerance, heterogeneity and openness. Wijegunaratne and Fernández (1998) consider that an environment supporting distributed applications with components that cooperate dynamically must have the following attributes: access and security, concurrency and maintenance of consistency, fault-tolerance (availability), heterogeneity and transparency, inter-process communication, naming, openness, scalability and resource sharing and management.

On the other hand, Coulouris, Dollimore and Kindberg (1994) indicate that the problems that an architect finds in the construction of a distributed system, arising specifically from the own nature of the system, are the following: naming, communication, software structure, workload allocation and consistency maintenance. Similarly, Forslund, Barry, Vines, Raj and Tiwary (1998) point the following as typical problems in the distribution: naming, reliability, availability, replication, fault-tolerance, transactional integrity, persistence, security, object mobility, heterogeneity, scalability and performance.

From our perspective, the set of terms indicated above to characterise the quality attributes of distributed applications fall into three different categories or points of view (fig. 2):

- *Intrinsic*: those non-functional requirements that derive directly from the own nature of the system, that is, those that are intrinsic to the distribution. In our case the following: concurrency, heterogeneity, communication, transparency and shared resources.
- *Specific*: non-functional requirements denoting a goal to reach, coming from the specification of the application to develop. In our case, they are: performance, reliability, scalability, extensibility, access and security, and dynamic reconfiguration.
- *Mechanisms*: those terms referencing mechanisms to support the requirements from both previous groups. In our case they are: naming, replication, object mobility or migration, load assignment or balancing, persistence, software structure, consistency maintenance, transactional integrity and fault tolerance.

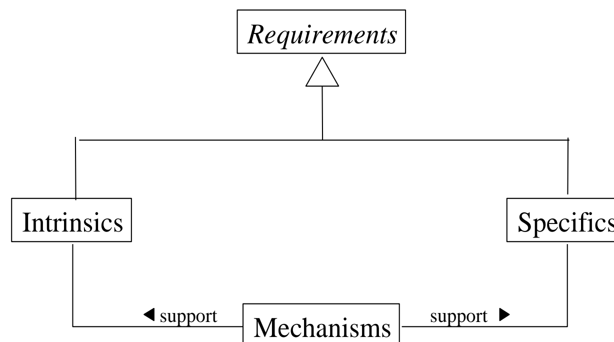

Fig. 2. Taxonomy of the quality attributes in the distributed applications.

Of course, we know that the previous allocation of terms to categories is not perfect. Some requirements classified as intrinsic, in specific circumstances, should not exist in the distributed application. For example, we can build a new application in which every element (hardware platform, operating system, communications mechanisms, programming languages, etc.) were homogeneous or an application that do not support the performance transparency.

On the other hand, the specific requirements depends on the type of the application. Each application has its particular requirements. For example, we can build an application without varying load and, therefore, it does not need the scalability requirement.

Finally, the mechanisms that a distributed application need in order to support the intrinsic and specific requirements will vary according these requirements.

Nevertheless, with this clasification we have tried to represent the generality, the attributes that usually characterize the distribute applications. We are aware of "the exception proves the rule".

Furthermore, we think that this classification throws light upon two important aspects related to the quality attributes of distributed applications:

1. It establishes a clear distinction among what quality attributes are and what mechanisms are.
2. It differentiates between those attributes that are always present in all distributed application (intrinsic) and those attributes whose existence depends on the specification of a particular application (specific).

4. Services to support the quality attributes

In this section we will study the services needed to support the quality attributes of a distributed application. We will also show how these services are associated to the classes included in Fig. 1 and to new classes that we will define.

4.1 Intrinsic quality attributes to distribution

Concurrency comes naturally in distributed systems due to the existence of several user processes executing in different computers. These processes access *shared resources* (hardware and

software) and/or update them concurrently generating a consistency problem in updates. To solve this problem we need a concurrency control service (CCS) that will also be responsible for supporting concurrency transparency (enables several processes to operate concurrently using shared information objects without interference between them). Besides, to be able to use a resource, the process must know about it. This is why we provide a naming service (NS) that will also support location transparency (enables information objects to be accessed without knowledge of their location).

Heterogeneity means that the system is formed by hardware and software elements, very likely heterogeneous (different hardware platforms, operating systems, compilers, communication protocols, etc.). This fact implies that the different elements must be interoperable. This problem can be solved by using a set of standards whose use hides these differences, such as the Interface Definition Language (IDL) or the Internet Inter-ORB Protocol (IIOP) of CORBA (OMG, 1999). The definition of the object interfaces using IDL and a compiler of this language that generates stubs and skeleton supports the access transparency (enables local and remote information objects to be accessed using identical operations).

Communication and synchronization protocols are in general related to application requirements such as performance and reliability. Emmerich (2000) describes communication from three orthogonal points of view:

1. Synchronization, that specifies different lock policies for the message passing for both who sends and who receives. These policies include: synchronous, oneway, deferred synchronous and asynchronous.
2. Multiplicity, that indicates how many processes participate in the communication. Three types can be distinguished: unicast, with a unique sender and a unique receiver; group request, where a sender passes the same message to a group of processes (using, for example, an events service); and multiple request, where different messages are sent to different receivers.
3. Reliability, that indicates the confidence degree of the client in which the server has executed the required service. For unicast requests the following degrees can be distinguished: exactly once, atomic, at least once, as much once and may be. For multicast requests the following forms can be distinguished: k-reliability, totally ordered and best effort.

Interprocess communication can produce cache coherence problems, but the same mechanism used to support the selected communication scheme must solve them. Therefore, we will define the basic communication service (BCS).

In terms of transparency, we already pointed out that the naming service can support *location transparency*. The *access transparency* is provided by the stubs and skeletons generated by a compiler of a neutral language. On the other hand, the concurrency control service takes charge of supporting the *concurrency transparency*. Support for *replication transparency* (enables multiple instances of information objects to be used to increase reliability and performance without knowledge of the replicas by users or applications programs) will be provided by a service implementing the operation to replicate an object. This operation may be conceptually modelled with the operation to copy an object (plus the operation to move it, if the replica will be located in a different node). Thus, we will define the life cycle service (LCS) of an object. The mechanism to support replication must consider the replica consistency problem. This service will also be responsible to support *migration transparency* (allows the movement of information objects within a system without affecting the operation of users or application programs) and *performance transparency* (allows the system to be reconfigured to improve performance as load vary). For the last one, the service must control load assignment and balancing in each node. *Failure transparency* is a consequence of fault tolerance, and the use of distributed transactions allows programmers to establish points from which the system recovers transparently from a failure. This service (LCS) must solve the problem of consistency when failures occur. Finally, support for *scale transparency* (allows the system and applications to expand in scale without change to the system structure or the application algorithms) will be provided by the system architecture and by the scalability of the used algorithms.

To summarize, in Table 1 we show the services needed to support each one of the intrinsic quality attributes to distribution.

Intrinsic quality attributes / services	CC S	NS	BC S	LC S
Concurrency and shared resources				
Heterogeneity	standards			
Communication				
Access transparency	standards			
Location transparency				
Concurrency transparency				
Replication transparency				
Failure transparency				
Migration transparency				
Performance transparency				
Scale transparency	software architecture			

Table 1. Services to support intrinsic quality attributes.

Fig. 3 shows how services defined in previous sections should be included in Fig. 1.

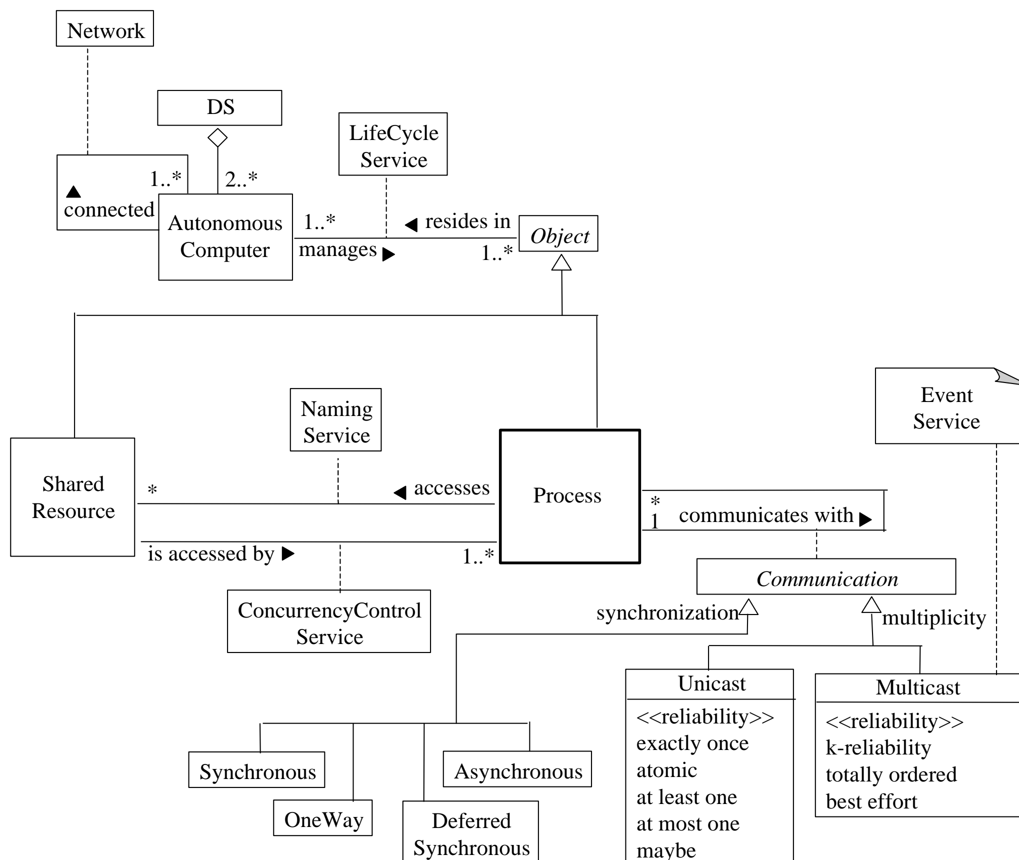

Fig. 3. Services to support the intrinsic quality attributes to distribution.

4.2 Application specific quality attributes

In this section we consider all those quality attributes that depend on the particular distributed application that we are going to build. As we will see, in general, these requirements do not need

any service to support them. They are more like a function of the efficiency of the services used to support the intrinsic quality attributes and the application architecture.

In general, application *performance* can be quantified. For example, we can specify the minimum number of transactions per minute that a server must support or the maximum response time allowed for a request. In a distributed environment, performance depends on aspects such as replication, load balancing/assignment, concurrency degree, selected communication mechanism, costs for consistency maintenance and efficiency of the naming mechanism

Reliability is the sum of correction and robustness, where correction refers to the system behaviour in specified cases and robustness refers to the system behaviour in non-specified cases (Meyer, 1997). For our analysis, we replace robustness by fault tolerance. The reliability degree of a system depends on aspects like the communication scheme used (see section 4.1) and fault tolerance.

The *scalability* requirement allows a load increase without changing the structure of the system or the application algorithms. Support for this requirement will be provided by the system architecture and the scalability of the application algorithms.

Extensibility is a requirement that depends on the publicity of the shared resources access interface. As we will see later, support for this requirement depends on the software structure of the application.

According to the application profile, the *access and security* requirement may be functional or non-functional. Security in a distributed system includes three main aspects (Wijegunaratne and Fernandez, 1998):

- Access control: for every resource and for every operation on that resource, identities of the allowed requesters should be known, so that every request can be checked.
- Authentication: for every request, the identity of the requester must be reliably ascertained.
- Auditing: the facility should exist to log every operation if desired.

Emmerich (2000) also includes the non-repudiation property in this non-functional requirement. The set of operations constituting this services will be defined in the class AccessAndSecurity (Fig. 4).

The *dynamic reconfiguration* requirement refers to the possibility of modifying the system architecture during execution time, creating, destroying or migrating its components. Thus, it is a requirement that the application architecture must support based on the life cycle service.

To summarize, in Table 2 we show the services needed to support each one of the application specific quality attributes.

Specific quality attributes / services	CCS	NS	BCS	LCS	AaS
Extensibility	standards				
Scalability	software architecture				
Dynamic reconfiguration				■	
Reliability			■	■	
Performance		■	■	■	
Access and security					■

Table 2. Services to support application specific quality attributes.

4.3 Mechanisms to support quality attributes

In this section we will describe the mechanisms somehow involved in the support of one or more quality attributes. Each mechanism will be supported by one or more services.

The *naming* mechanism supports the location transparency and serves any other component that requires access to shared resources. The naming service must be easily scalable and use an efficient translation method. As we mentioned in section 4.1, we will support this mechanism through a naming service.

The *replication* and *migration* mechanisms may be supported by operations included in the life cycle service. As we said before, the efficiency of these mechanisms affects performance, reliability and dynamic reconfiguration of the application.

The *load assignment* mechanism allows us to distribute processing, communications and resources to optimize performance in case of load variations. Since it consists in migrating or replicating objects, this mechanism can be supported by the life cycle operations together with a load assignment policy that measures load in every node and distributes adequately the computational costs.

The *persistence* mechanism allows an object to live longer than the process using it. It is needed to have a persistence service (PES) with operations to store and load the status of an object. This service will require the object to know how to externalize/internalize its status. Therefore, we also need an externalization service.

The *software structure* is the mechanism supporting the extensibility requirement. We have to design components with well defined interfaces and structure the system in such a way that when new services are included, they are totally integrated with the existing ones without duplicating service elements already present (Coulouris, Dollimore and Kindberg, 1994). This mechanism is based on the definition of a neutral interface language and in the publication of existing software repositories.

In distributed environments, we need some mechanism responsible for *consistency maintenance* in all its forms. We already saw that the concurrency control service is responsible for maintaining consistency in updates, that the life cycle service must take care of maintaining replica and failure consistency, and that the mechanism chosen to support the communication scheme (among processes) must take care of the cache coherence. Coulouris, Dollimore and Kindberg (1994) distinguish two more types of consistency: in clocks and in the user interface. To achieve the first one we need a time service (TIS). For the second one, the only thing we can do is minimizing the interface updating time.

Transactional integrity refers to the maintenance of the ACID properties of a distributed transaction. For transaction management, we need a transaction service (TRS). Observe that to support the isolation property, the transaction manager must rely on a concurrency control service. On the other hand, the durability property needs a persistence service.

Fault tolerance may be achieved through hardware and/or software redundancy. Sommerville (2001) indicates that "the most commonly used hardware fault-tolerance technique is based around the notion of triple-modular redundancy (TMR). The hardware unit is replicated three (or sometimes more) times. The output from each unit is compared. If one of the units fails and does not produce the same output as the other units, its output is ignored". For software fault tolerance, Sommerville (2001) cites techniques N-version programming and Recovery blocks. Related to fault tolerance is the availability characteristic, understanding it likes a measure of the percentage of time that the system is ready to use. Like we indicated earlier, fault tolerance can be supported by replication and directly affects system performance and reliability. To replicate a component, it would be enough to use the operations copy and move (if necessary) of the life cycle service.

To summarize, in Table 3 we show the services needed to support each one of the mechanisms.

Mechanisms / services	CC S	NS	BC S	LC S	Aa S	PE S	TIS	TR S
Naming								
Replication								
Migration								
Load assignment								
Persistence								
Software structure	standards							
Update consistency								
Failure consistency								
Replica consistency								
Cache coherence								
Clocks consistency								
Transactional integrity								
Fault tolerance								

Table 3. Services to support the mechanisms.

In Fig. 4 we represent, through a class diagram, the set of services required to support the quality attributes of a distributed application.

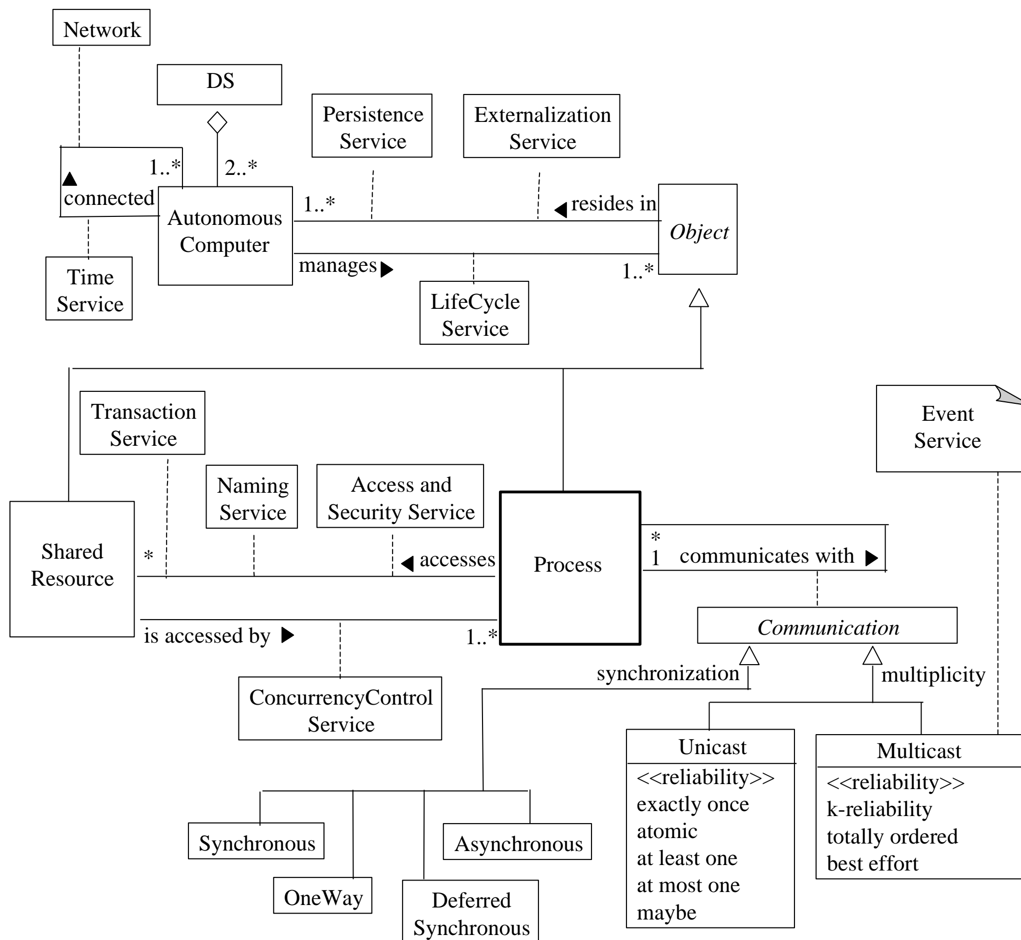

Fig. 4. Services to support the quality attributes of a distributed application.

5. Conclusions and future work

In this work we have established a taxonomy of the quality attributes of distributed applications where the requirements were classified as: intrinsic, specifics, and support mechanisms. We established the services needed to support quality attributes such as concurrency, communication, different forms of transparency and scalability.

We are studying the orthogonality of the quality attributes in the distributed applications domain. From this study, we expect to obtain a minimal set of services to support those attributes. After determining that minimal set of services, we will study which elements of a software architecture have to support those services; that is, whether those services can be assigned uniquely to the connectors or whether the components are necessary to support them. With all this we hope to describe a reference architecture for distributed applications that, among other features, eliminates the implicit architectures problem generated by the middlewares.

References

Bass, L., Clements, P. y Kazman, R. (1998). *Software architecture in practice*. Reading, MA: Addison-Wesley.

Coulouris, G., Dollimore, J. and Kindberg, T. (1994). *Distributed systems, concepts and design* (2nd ed.). Reading, MA: Addison-Wesley.

Emmerich, W. (2000). *Engineering distributed objects*. West Sussex, England: John Wiley & Sons.

Forslund, D., Barry, T., Vines, D., Raj, R. y Tiwary, A. (1998, october). Building distributed systems [panel]. *Proceedings of Object-Oriented Programming, Systems, Languages and Applications* (pp. 412-416). Vancouver, Canadá: ACM.

Kotonya, G. y Sommerville, I. (1998). *Requeriments engineering: processes and techniques*. West Sussex, England: John Wiley & Sons.

Meyer, B. (1997). *Object-oriented software construction* (2nd ed.). NJ: Prentice-Hall.

OMG (1999). *The common object request broker: Architecture and specification*. Author.

Pressman, R.S. (2001). *Software engineering: A practitioners approach* (5th ed.). London: McGraw-Hill.

Rumbaugh, J., Jacobson, I. and Booch, G. (1999). *The unified modeling language reference manual*. Reading, MA: Addison-Wesley.

Sommerville, I. (2001). *Software engineering* (6th ed.). Harlow: Addison-Wesley.

Wijegunaratne, I. and Fernandez, G. (1998). *Distributed applications engineering*. Great Britain: Springer-Verlag.