Author(s): Lago, Patricia; Falcarin, Paolo.
Article title: UML requirements for Distributed Software Architectures
Year of publication: 2001
Citation: Lago, P. and Falcarin, P. (2001) 'UML Requirements for Distributed Software Architectures', in Proceedings of the 1st International Workshop on Describing Software Architecture with UML, Toronto, Canada, May 2001, pp. 27-30.

UML requirements for Distributed Software Architectures

Patricia Lago
Dipartimento di Automatica e Informatica
Politecnico di Torino
Corso Duca degli Abruzzi 24
I-10129 Torino, Italy
+39 011564.7008
Patricia.Lago@polito.it

Paolo Falcarin
Dipartimento di Automatica e Informatica
Politecnico di Torino
Corso Duca degli Abruzzi 24
I-10129 Torino, Italy
+39 011564.7091
Paolo.Falcarin@polito.it

ABSTRACT
The use in various projects of UML as the modeling notation for representing software systems, revealed the need for additional constructs and architectural views, especially in the field of distributed software architectures. This article identifies a list of requirements toward UML, which we find appropriate especially in its current standardization stage.

Keywords
Unified Modeling Language, Object Management Group.

1 INTRODUCTION

The intensive use of OMT and thereafter UML as the modeling notation for software development throughout various projects, led to the identification of a set of lacks in the UML notation, and to the definition of associated requirements for UML extensions. Experience, on which this article is based, has been acquired in both didactic small-scale projects (e.g. [3][2]), and large-scale projects, in which development was both centralized and distributed among multiple remote developers and development partners. Throughout the years, difficulties and limitations have been identified and associated extensions have been applied, especially because large-scale projects needed a consensual diagramming notation to model detailed software aspects. References to some of these projects results can be found in [6][5][1].

2 REQUIREMENTS

Class Diagram

#1. Different diagrams modeling Information and Computational classes

Of particular relevance in a distributed system, is the twofold role of classes, which can model either information distributed over networked sites, or components (and computational objects) that manipulate that information.

Accordingly, UML should support two views over a system, the information view and the computational view, as well as the relationships between the two views, which are not straightforward, depending on how information is distributed over computational elements [4].

The *information class diagram* should describe a system in terms of the managed information elements, their semantics, and their logical relationships.

The *computational class diagram* should describe a system as a collection of interacting components that maintain a set of information elements, provide an interface for their manipulation, and interact with other components to achieve system functionality. To this aim, the following aspects are essential in the computational class diagram:

- System components, i.e. the atomic modules aggregating a collection of computational classes. Component representation should identify both the internal and external structure, in a graphical compact notation.
- Exported interfaces, i.e. the interfaces that a component offers to external distributed invocations.
- Internal interfaces, i.e. the interfaces that are offered by the classes the component encapsulates, but that are not externally visible.
- Component associations, i.e. relationships between different components, and (inside a component) between internal classes.

The clear and immediate identification of distributed interfaces and interactions (also from a plain graphical representation) provides the starting point for stating the structure of each component, its usage, the number and type of interfaces it should export along with their usage from external clients.

#2. Internal details of a component

Each component should clearly identify: remote interfaces from local language-specific interfaces; which internal objects export which remote interfaces; interaction relationships between internal objects; which internal objects invoke external component interfaces. Also, and especially in distributed systems, it is important to differentiate static objects (instantiated during component

creation and active for the whole component lifecycle) from dynamic objects (whose number varies according to individual execution scenarios).

As this requirement directly addresses the structure of a component, it could be considered in the extension of the component diagram too, by providing two abstraction views: the inter-component view and the intra-component one.

For example, Figure 1 proposes a compact representation of the computational diagram, for a simple but still complete generic component named `Server X`. We can identify at once that the component exports four distributed interfaces (named with prefix `ii_`), and that it controls an external database. Concerning the internal structure, it is also clear that remote interfaces are all implemented by one internal object (`X manager`), which controls objects of both classes `X_managed_object` and `db_proxy`, by invoking their (local) interfaces.

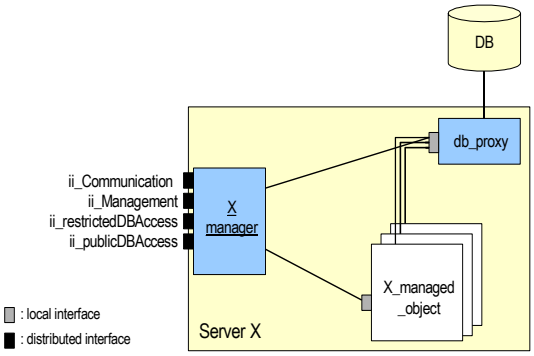

Figure 1. Computational diagram

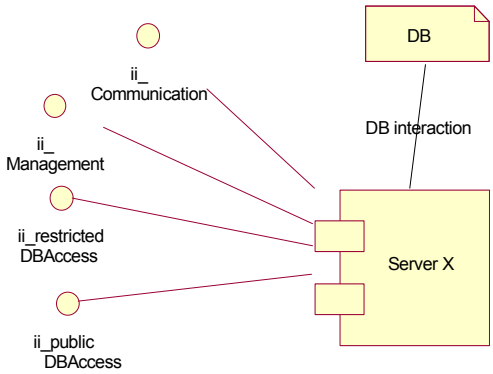

Figure 2. UML Component diagram

Further, internal objects are graphically represented in a different way: dark objects indicate the ones that exist in a single instance, and whose lifecycle corresponds to the lifecycle of the component itself, and white objects represent transient instances, whose number varies during the component lifecycle.

At last, the name of factory objects (i.e. a special kind of static objects exporting the interface that implements component creation and deletion) is underlined.

#3. Behavioral specification

The loose matching between software specification and implementation is a problem, especially in distributed software architectures where multiple components are weakly specified, and multiple teams carry out their development. A key aspect to solve this problem is to provide behavioral specification.

Behavioral specification can be defined as the formal description of what is supposed to happen when software executes. Object Constraint Language (OCL) included in UML specification is a complete formal language used to set invariants, pre- and post-conditions related to class attributes, methods, and associations.

Current practice manages these data only as another form of documentation, often ignored for implementation purposes. Instead, OCL constraints could become a more powerful help if supplementary UML specifications would be introduced in order to translate these data into implementation constructs. Translation from UML notation to common programming languages is defined by UML mapping specifications; for example UML to IDL conversion could consider OCL conditions and copying them as comments in IDL code. Accordingly, a future special IDL compiler (e.g. for C++ or Java) could translate these data inside implementation classes.

Alternatively, the mapping from UML to the preferred programming language could be extended in order to translate OCL conditions in related skeleton code of class implementation, e.g. containing methods that verify correctness of constraints, possibly raising exceptions when not valid.

Component Diagram

#4. External structure and interactions of components

The component diagram, being the representation of a software system in terms of components, should model in a compact style the external (remote) interfaces offered by each component, the clients of each interface as well as the used external interfaces of other components.

The overall objective of this diagram (providing what has been called in the previous section inter-component view) should be the identification of potentially distributed

interactions, starting point for deployment analysis.

As an example, we modeled in Figure 2 the component diagram for Server X, already represented in Figure 1. We can observe that (although representing internal details too) our computational diagram results more compact and readable.

The example is simple and represents a single component, but if we think of a computational model representing multiple components as well as their inter-relationships, we can easily imagine how complexity explodes with a consequent loss in readability. Instead, the component diagram with mixed notation could just focus on these inter-component interactions, whereas the computational class diagram could detail intra-component decomposition and internal endpoints of inter-relationships.

Sequence diagram

#5. Representation of composite objects and multiple interfaces

In a complex system, the invocation toward a component cannot be simply represented as an arrow toward a component instance represented as an atomic object. It should be clearly represented which interface of that component is invoked (whenever components offer multiple interfaces), and which component object exports that interface (whenever the component has a composite structure). This representation should be compact, to easily identify the component structure.

Also, it should be possible to hide/show this component structure, to build diagrams at different abstraction levels (see requirement below).

#6. Vertical modularization

As a system becomes complex, there is the need to model interactions on two abstraction levels, that is outside components (inter-component) and inside a component (intra-component). This allows to have first an overview about the complete set of interactions needed to complete collaborative system functionality, and then to detail how each invocation toward each involved component is actually implemented internally.

Intra-component sequence diagrams should model external (invoking and invoked) components as atomic objects, whereas it should decomposed intra-component interactions for the analyzed component only. This refinement permits to identify which internal objects invoke external interfaces, and how the functionality is internally served.

#7. Horizontal modularization

When a diagram has to represent a complex execution scenario made up of a large number of interactions, it is convenient to split the diagram into multiple diagrams, each representing a group of invocations. To maintain readability, invocations can come from "nowhere" and go "outside" the diagram, to mean that these invocations belong to a previous/successive sub-diagram.

Also, it is often the case that a group of invocations are common to multiple scenarios (e.g. an authentication procedure common to multiple service scenarios). Instead of repeating the same invocation in all diagrams, these could be represented in a separated general sub-diagram, and then referred into the diagrams that include it.

#8. Return parameters on return interaction arrows

Return arrows are very important to model the complete cycle of an invoked operation. Even more important is the explicit representation of the list of parameters that are returned to the client component. In particular, these define both the output parameters of remote interface methods, and the exceptions raised during supported failure scenarios (as to be specified in OMG/IDL [7]).

#9. Recurring interaction patterns should be modeled at class level

Each medium-to-large-size software system includes a set of management scenarios that require careful specification, especially if invocations cross network lines. Examples are system boot with component creation, registration and reference exchange, component deletion or partial failure recovery. To this aim, common management scenarios should be represented by class-level sequence diagrams, to make up a recurring pattern.

Deployment diagram

Deployment diagrams show the run-time configuration of software systems. To this aim, they should fulfill the following requirements:

#10. It should map the computational class diagrams onto a modeled execution environment

This aspect is crucial for successful deployment and on-field test, as it permits to decide which components can be deployed on remote nodes or local machines, which interactions take place on networked lines, which interfaces are to be registered on a middleware platform, which components are duplicated for failure resiliency.

Furthermore, the deployment of remote interfaces over distributed nodes highlights which system components need extensions to implement security mechanisms.

#11. Verification support for standardization and business issues

The deployment model should be customizable according to a defined business model, i.e. the identification of boundaries between business administrative domains. This extension would give large support in the identification of cross-domain interactions, to check compliance to business model reference points [8], and to inter-domain standards at interface level.

In summary, fulfillment of this requirement would give support to: (1) directly elicit distributed communication

implementing cross-domain reference points; (2) check compliance to standards (whenever applicable); (3) carry out system component analysis.

3 CONCLUSIONS AND FURTHER WORK

This article identifies a list of requirements that should be considered in the definition of the UML for the representation of software systems. In particular, the work here presented is the result of more than five years working in research projects developing complex object-oriented software architectures, as described in [4]. The work just focuses on the main UML diagrams that are of particular relevance for the representation of a complex system, and that represent the minimal set of diagrams, common to any type of software. Of course, additional requirements could be identified for domain specific systems. For instance, distributed systems export a set of remote/distributed interfaces that are usually defined in OMG/IDL (Interface Definition Language) and then mapped on the particular implementation language and distribution mechanisms. IDL specification that are usually considered in the starting stage of implementation, are instead a powerful mean during specification and early design, especially in this technological era in which compliance to standards is often specified by means of IDL interfaces that could directly be adopted to guide the design process. In this respect, IDL specifications (or more in general interface specification) could be included as new diagramming constructs.

Further, for systems with relevant dynamic complexity, the use of Statechart diagrams should be extended for dynamic verification purposes, and diagram elements should be considered in the mapping toward programming languages, for the generation of source code skeletons (e.g. like suggested in requirement #3).

REFERENCES

[1] Canal, G., Lago, P., Integration of Commercial Internet Applications in a TINA Environment. In *Proc. TINA'99* (Oahu, HI, Apr. 1999), IEEE Computer Society Press, 2-13.

[2] Jaccheri, M.L., Lago, P., Applying software process models and improvement in academic setting. In *Proc. CSEE&T'97* (Virginia Beach VA, Apr. 1997), IEEE Computer Society Press, 13-27.

[3] Jaccheri, M.L., Lago, P., How Project-based Courses face the Challenge of educating Software Engineers. In *Proc. SCI'98-ISAS'98*, (Orlando FL, Jul. 1998), 377-385.

[4] Lago, P. Rendering Distributed Systems in UML. In *Unified Modeling Language: Systems Analysis, Design, and Development Issues*, K. Siau and T. Halpin (Eds.), Idea Group Publishing, ISBN 1-930708-05-X, 2001, pages 350.

[5] Lago, P., et al., TINA Service Architecture: From Specification to Implementation. In *Proc. TINA'97* (Santiago Chile, Nov. 1997), IEEE Computer Society Press, 174-183.

[6] Lago, P., Licciardi, C.A., Canal, G., Andreetto, A., An architecture for IN-Internet hybrid services. In *Computer Networks Journal*, Special Issue on Intelligent Networks and Internet Convergence, T. Magedanz, H. Rudin, I. Akyildiz (Eds.), Elsevier, Vol. 35(5), April 2001, 537-549.

[7] Object Management Group Web Site. On-line at http://www.omg.org.

[8] TINA Consortium, TINA Business Model and reference Points, TINA-C Baseline, v4.0, May 1997. On-line at http://www.tinac.org.