Architectural Views for Designing Relationships in Distributed Systems

Anca Daniela IONITA¹, Jacky ESTUBLIER²
¹University Politehnica of Bucharest
Splatul Independentei 313, 060042, Bucharest, Romania
Anca.Ionita @ mag.pub.ro
²LIG-IMAG, 220, rue de la Chimie BP53 38041 Grenoble Cedex 9, France
Jacky.Estublier @ imag.fr

Abstract — The paper studies the detailed design of relationships between concepts, taking into account four architectural views: logical, process, implementation and deployment. The classical patterns, applied if one considers the conceptual relationships from the logical view, are not enough for distributed applications, because the related elements may pertain to different computing nodes. More criteria should be used for the detailed design, like the technology used for communication paths - e.g. for Service Oriented Architectures - or the rights to modify the subsystems containing the related elements - if the distributed system is based on composing reusable assets. Moreover, the object oriented approach may need to be leveraged by a design based on aspects, for grasping crosscutting issues.

Index Terms — Aspect-Oriented Software Development, Design methodology, Distributed systems, Object oriented methods, Service Oriented Architectures

I. INTRODUCTION

The relationships between concepts are essential in modelling languages, as in Entity Relationship (ER) Diagrams used for databases, in the class diagrams from Unified Modeling Language (UML) and in Domain Specific Languages (DSLs). Taken the UML classification of relationships, generalization has a direct mapping on object oriented languages, but associations and aggregations may be designed in various ways. The difference between them is not structural - it stands in the life cycle of the related elements [1]. Qualified associations, aggregations and compositions may be decomposed into simple associations, enriched with OCL constraints [2], transforming complex UML concepts into basic ones, for easier transformations to executable code. Besides the associations expressing structural relationships, there are others, resulting from communication links, expressing a behavioural relationship [3]. Normally, they should not appear in class diagrams [4], but the related objects may pertain to different subsystems, as in the new law of Demeter [5] for concerns, so they need a structural support for interacting.

The paper investigates the case of associations, for situations where they cross subsystem boundaries, in distributed systems. For doing this, we investigate the implications of various architectural views, which are created for modelling the system from the viewpoints of various stakeholders, as defined in the standards IEEE Std 1471-2000 and ISO/IEC 42010:2007. Specific view models were defined for distributed architectures (like RM-ODP, the Reference Model for Open Distributed Processing [6], adopted by ISO) and for Service Oriented Architectures (SOA) (like the Reference Architecture for SOA [7], from OASIS). For the purpose of this study, we used the variant of the “4+1” View Model that binds the Philippe Kruchten’s views [8] on UML notation [9]. The logical view represents the structural elements, the process view shows the sequencing of activities, the implementation view presents the decomposition in subsystems and the deployment view outlines the distribution on physical nodes. The multiple views of the system have an important role in its description and documentation for various stakeholders (see Table 1).

The study starts with the classical, object-oriented patterns for designing relationships (Chapter II) and continues with two kinds of relationships:
- The former - relating concepts across distributed nodes, for example with Web services (see Chapter III);
- The latter - relating concepts that belong to immutable, reusable assets, using aspect oriented design (see Chapter IV).

Then, two case studies are discussed as real-world examples: a transnational e-Business system developed in LD-CAST project [10] (Chapter V) and the horizontal relationships introduced in Mélusine [11] (Chapter VI).

<table>
<thead>
<tr>
<th>Views</th>
<th>Characteristic UML Diagrams</th>
<th>Stakeholders</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logical</td>
<td>Class</td>
<td>End users</td>
</tr>
<tr>
<td></td>
<td>Package</td>
<td></td>
</tr>
<tr>
<td></td>
<td>State Machine</td>
<td></td>
</tr>
<tr>
<td>Process</td>
<td>Sequence</td>
<td>System integrators</td>
</tr>
<tr>
<td></td>
<td>Communication Activity</td>
<td></td>
</tr>
<tr>
<td>Implementation (Dev)</td>
<td>Component</td>
<td>Programmers</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Deployment (Phys)</td>
<td>Deployment</td>
<td>System engineers</td>
</tr>
<tr>
<td>Use Case</td>
<td>Use Case Activity</td>
<td>All (Connecting the 4 views)</td>
</tr>
</tbody>
</table>

II. CLASSICAL PATTERNS FOR DESIGNING RELATIONSHIPS

The design of associations depends on their properties, specified in the Logical View with the UML class diagrams, like in Figure 1. Navigability and multiplicity are the most important, involving structural differences; for example, for unidirectional, N-ary ordered associations, one introduces a new attribute that represents a list of references to the associated objects, plus various operations that manipulate them (see Figure 2).
Moreover, there are also behavioral aspects that have their influence, like the moment when the source and destination instances are linked. The binding may be performed:
- **early**, creating the link immediately after the instantiation of the source, in the ConceptA constructors;
- **late**, creating the link when its navigation is demanded, by calling an operation like setAssociatedObject at the right moment; this is specified in UML in the sequence diagram – pertaining to the Process View.

Moreover, the moment of creating the associated object may also be influenced (as described in the sequence diagrams); it might have existed before instantiating the relationship, or may be created when the link is requested. Still, the life cycle should not be influenced more than this; the associated object should evolve independently, if one respects the nature of the relationship. This paper is not concerned with the life cycle for aggregations.

From the non-functional point of view, the link persistency may also be important; generally, information related to association instances are saved together with the related objects. However, one should pay attention to the association classes, which should be separately considered for serialization, or for creating new table entries in databases.

III. DESIGNING RELATIONSHIPS IN DISTRIBUTED SYSTEMS

Besides the logical information grasped in the class diagram, the Implementation (Development) View may reveal that the related concepts pertain to different subsystems, so they may be linked through an appropriated interface, as depicted in Figure 3.

The Deployment View comes along with the possibility that the two concepts are actually implemented on different nodes, imposing a proper design for the connectors, in respect with the adopted standards, like HTTP and SOAP in Figure 4, where one supposed that the communication is performed with Web services. For modeling the service in UML, one may use the profile proposed in [14], where SubsystemB was considered a provider of inner services, the interface became a core service - supporting functionality for implementing distributed links - and the service exchanges information tokens through a message (see Figure 5). A standard entitled “UML Profile and Metamodel for Services” is in course to be adapted by Object Management Group (OMG).

The paper will exemplify this situation on the LD-CAST prototype, which was developed to support local development cooperation actions using a Service Oriented Architecture (SOA) and semantic technologies (Chapter V).

IV. RELATING CONCEPTS FROM IMMUTABLE SUBSYSTEMS

The evolution of tools is towards growing in size, complexity and functionality. Along with the increasing demands for complex requirements, software industry has developed larger and larger systems, which are difficult to build, maintain, learn and customize. However, there is also an opposite trend, trying to build applications by allowing pieces of existing software to interoperate. The purpose is to build new applications with rich functionalities, while not increasing the dimensions of individual tools, but making them inter-operate with either new or existing programs. This alternative trend places a strong emphasis on reusability of non-homogenous artefacts, like COTS (Commercial Off The Shelf), legacy software or any other products available on the market, designed to interoperate or not.
Figure 6. Defining aspect oriented relationships.

This chapter analyzes the design decisions entailed by the constraints of this second trend: (i) one does not have access to modify the code of the composed subsystems, and (ii) conceptual relationships should be established between elements situated in immutable pieces of code. A pattern for overcoming this constraint is given by Aspect Oriented Software Development (AOSD) [15], which is able to extend the structure and modify the behaviour without touching the original code. In this case, the relationships cannot be treated with the classical, object oriented approach, presented in Chapter II, because they become crosscutting concerns, whose life cycle has to be managed separately.

Figure 6 presents two subsystems that cannot be modified, but expose the interfaces of their factories for objects instantiated from ConceptA and ConceptB respectively. To design a relationship between the two classes, one has to create an aspect that incorporates all the code related to its structure, life cycle management and interactions performed across its links.

The aspect can prepare the relationship structure by reifying the pattern from the object oriented design and declaring inter-type members owned by ConceptA, like:

```
private ConceptB ConceptA.associatedObject;
public ConceptA.setAssociatedObject(ConceptB oB) {};
```

Alternatively, one may declare a new interface, ConceptABFactory, which extends ConceptAFactory and creates objects of type ConceptAB, an extension of ConceptA. An example of this approach is given in Chapter VI. This pattern should be carefully used, as it disobeys the encapsulation principle, adopted when designing the original subsystems.

V. CASE STUDY 1: RELATIONSHIPS IN AN E-BUSINESS PLATFORM BASED ON SOA

This chapter analyzes relationships between concepts in a system for facilitating business cooperation, designed and developed in LD-CAST STREP project. From the logical point of view, the main concept is BusinessService, which may be supplied through various business processes, as depicted in the UML class diagram from Figure 7. The complex business services are supplied by orchestrating Web services delivered by various suppliers across Europe, like chambers of commerce or financial institutions.

Before being able to present such a service to the End-User, at design time, the system has to represent the Business Process Models (BP_Model) that describe it, with the BP Modeler subsystem, which is deployed on another platform and uses a language that is easy to understand for business experts. After that, one has to define the correspondent Abstract Workflow (AbstractWF) - hosted by the Process Execution Manager (PEM) and written in an executable language (BPEL).

At run time, the End-User has to make some choices for requesting the appropriate business service and, according to them, one determines the concrete workflow (ConcreteWF) that is going to be executed on PEM, according to the configuration of Web services (WS_Configuration) that best fits the user preferences. The relationships between these concepts appear in the Logical View, as seen in Figure 7, but the problem is that the related concepts are actually implemented and persisted in different subsystems (see Table II). Moreover, these subsystems are deployed on different nodes (Figure 8). The solution adopted in LD-CAST was to define Web services for the communication between distributed nodes, because the system was already using services for implementing workflow activities.

<table>
<thead>
<tr>
<th>TABLE II. RELATED CONCEPTS IN LD-CAST</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConceptA</td>
</tr>
<tr>
<td>------------------</td>
</tr>
<tr>
<td>Business Service</td>
</tr>
<tr>
<td>Process</td>
</tr>
<tr>
<td>BP Model</td>
</tr>
<tr>
<td>Requested BS</td>
</tr>
<tr>
<td>Concrete WF</td>
</tr>
</tbody>
</table>

Figure 7. Logical View of LD-CAST.

Figure 8. Deployment View of LD-CAST.
VI. CASE STUDY 2: ASPECTS FOR RELATING IMMUTABLE DOMAINS

This chapter analyzes relationships between concepts pertaining to different models, in a federative architecture based on composing subsystems called domains, whose execution is driven by specific metamodels and is supported by Mélusine platform [11]. A complex application may pertain to a composite domain, characterized by several models (one for each sub-domain) connected by inter-domain relationships.

This case study is based on RichProduct, a domain obtained by composing the autonomous domains Product and Document, while preserving their original implementation. The Product domain can manage versions of products characterized by typed attributes; these products are “empty” - they do not contain any file to describe them better. The Document domain manages versions of documents and stores them in a remote repository. Documents may be files or directories, and there may be an exclusion pattern for ignoring certain files in a directory.

The idea behind RichProduct domain was to have a versioning system for complex, “rich” elements, characterized by some attributes (managed by Product domain) and by supplementary information stored in one or more files (managed by Document domain). If Product domain alone rather corresponds to a PDM (Product Data Management) tool [16], RichProduct - obtained through its composition with Document – may be used in many other complex systems, e.g. for document management or for software configuration management. For this purpose, one has to establish inter-domain relationships between the sub-domains correspondent concepts, as seen in Table III.

<table>
<thead>
<tr>
<th>ConceptA</th>
<th>Sub-domain</th>
<th>ConceptB</th>
<th>Sub-domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product</td>
<td>Document</td>
<td>Revision</td>
<td>Document</td>
</tr>
<tr>
<td>Branch</td>
<td>Product</td>
<td>Revision</td>
<td>Document</td>
</tr>
</tbody>
</table>

Table III. Related Concepts in Rich Product

The constraint imposed by the federative methodology [17] was to keep unchanged the code for Product and Document domains. The inter-domain relationships between them have been designed as aspects that extend the classes of Product domain, in the manner presented in Chapter IV. The concepts RichProduct, RichBranch and RichRevision from RichProduct domain, described in the class diagram from Figure 9, are extensions of the classes Product, Branch and Revision from Product domain. For example, RichProduct is an extension of Product class from Product domain, such as to have an association with the Document class from the Document domain. RichProductDomain, allowing the creation of products with associated documents, is also an extension of the product factory: ProductDomain.

An example of AspectJ code for the method createProductWithContent, which creates objects of type RichProduct, is presented in Appendix A; exception treatment has been removed for outlining the paper ideas. The solution takes into account that the documents are hosted by a different node (see Figure 10). It was considered that Product domain is active, because its components are interactive and are influenced by non deterministic actors, like humans. On the contrary, Document domain was considered passive, only performing actions requested by other domains; therefore, its concepts are hidden to the user of the composite domain. The lifecycle of documents, with their branches and revisions, is under the complete control of products, with their respective revision tree. Thus, as observed in the example, the destination life cycle is dependent on the source one. Along with the methods for managing relationships lifecycle, the aspects may include other methods - introduced for designing interactions - as also performed by Blay [18], or by Lieberher, with aspectual collaborations [19].

VII. CONCLUSION

Designing relationships between concepts is a complex task, which has to take into account multiple issues: (i) the structural elements shown in the Logical View, (ii) the behavior imposed by the type of relationship, or by the description from the Process View, (iii) the Implementation View, which may reveal that concepts are managed by different subsystems, (iv) the Deployment View, if the system is distributed over several computing nodes and, last but not least, (v) the possibility to modify the code correspondent to these concepts. Experience showed that an object oriented approach may not be enough in this last situation, but an aspect oriented design in necessary.

APPENDIX A

Relationship Implemented with AspectJ

```java
public aspect RelH_Product_Document extends FedeAbstractAspect {
  public RichProduct ProductDom.createProductWithContent
      (ProductType productType, Map attributesMap, String host, URI directory, Pattern exclusion)
      {
        Product product = createProduct( productType, attributesMap );
      }
```
DocumentDom documentDom = (DocumentDom) MelusineCore.getDomainRoot (DocumentDom.NAME);
DocumentType documentType = getDocumentType (product.getProductType(), documentDom);
Document document = documentDom.createDocument (documentType, product.getIdentificator(), host, directory, exclusion);
product.link(document);
return (RichProduct) product;

ACKNOWLEDGMENTS

Special thanks to the LD-CAST partners for the experience gained during the project, which inspired the first case study of the paper. This work was supported in part by CNCSIS -UEFISCSU, project number PNII - IDEI 1238/2008.

REFERENCES