Subsystem Design Guidelines for Extensible General-Purpose Software

Paul Grefen and Roel Wieringa
Computer Science Department
University of Twente, The Netherlands
Phone +31-53-4894283
{grefen,roelw}@cs.utwente.nl

1. ABSTRACT

We discuss subsystem design for extensible general-purpose information systems. We extract guidelines from a case study of the redesign and extension of an advanced workflow management system and place them into the context of existing software engineering research. Key aspect is the distinction between essential and physical architectures, related to software clustering and distribution.

1.1 Keywords

Architecture design, subsystem design, general-purpose software

2. INTRODUCTION

Guidelines for subsystem design of general-purpose software are rare. Structured and object-oriented design methods mostly ignore the subsystem level [16, 10], and those authors who do discuss this level, confine themselves to frequently identified subsystems [1, 5, 11]. None of these authors discuss subsystem design guidelines for general-purpose systems. The field of patterns and software architecture mostly focuses on the programming language level [6, 2]. In addition, many of these approaches, especially the structured and object-oriented ones, discuss ways to *represent* architectural designs rather than ways to arrive at good designs. This syntactic orientation sometimes appears as a preoccupation with a particular programming language, such as Ada [12] or C++. Especially in situations where information systems are complex, e.g. because of advanced functionality, distribution, or interoperability, subsystem design guidelines are required to arrive at well-structured systems.

In this position paper, we discuss subsystem-level design guidelines that were used in the development of the WIDE prototype commercial workflow management system [3, 4, 7]. After summarizing related work in subsystem-level design in Section 2, we present the major design decisions made in the WIDE system in Section 3. In Section 4, we discuss the lessons learned of the project.

3. RELATED WORK

A *domain* can be defined as a "separate real, hypothetical, or abstract world inhabited by a distinct set of objects that behave according to the rules and policies characteristic of a domain" [11]. They recommend defining a subsystem for each domain of the system and give the following examples of domains: the user interface domain, the subject domain, the database domain and the operating system domain. Additional examples are the fault tolerance domain, alarm handling, hardware control and the device domain [1]. Although these examples are useful, they do not help clarifying the concept of domain. The concept of a separate world used in the definition of the domain concept is at best metaphorical, and it is hard to see how a set of objects could be anything but distinct. Eliminating these vague concepts from the definition, we are left with the concept of a domain as a part of the world for which there are characteristic rules and policies. This still leaves considerable freedom in deciding what is and what is not a domain. Moreover, it is not clear why the examples given are good domains. Defining a subsystem for all of these domains may lead to an unnecessarily complex system architecture.

Two subsystem partitioning criteria are [5]: the presence of aggregate objects may motivate the definition of a subsystem, and we may also identify subsystems that correspond to a set of external entities, data stores, or control objects. Examples of subsystems given are real-time control, real-time coordination, data collection and data analysis subsystems, and servers. Here too, one may recognize the idea of domains, but again it is not clear why one should identify these subsystems and not others. In addition, the design criteria are stated in terms of the elements of data flow diagrams, which illustrates the syntactic orientation of these guidelines.

What is needed in the design or complex systems, is an understanding of the reasoning behind subsystem design, so that this reasoning can be applied in different cases with possibly different results. We also observe that a general-

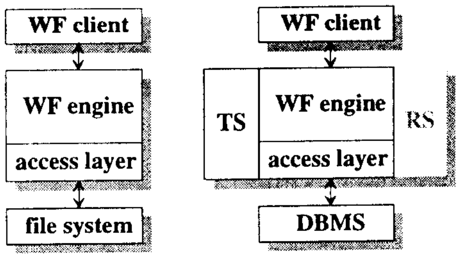

Figure 1 a, b: physical FORO architecture and overall WIDE architecture

purpose system may be used in many different contexts, so that it is less apparent what the application domain of the system is. A more general point of view is thus necessary. In this paper, we make a first step into this direction.

Another issue that we want to point out is the division of work between functional, object-oriented decomposition and distribution design. In [12, 13], a functional decomposition is made first that is independent of the distribution of the system, then functions are grouped into sequential processes (Ada tasks), and finally these processes are aggregated into concurrent objects that can be allocated to physical resources in a distributed system. By contrast, [1] starts by identifying concurrent objects in terms of the application domain and later group these into sequential processes (threads). These methods are divided on the issue of whether to use functional decomposition or decomposition based upon the application domain, and on the issue whether to partition into sequential processes before or after identifying objects. We return to this issue after the design choices in the WIDE system are discussed.

4. DESIGN OF THE WIDE WFMS

The WIDE project aims at complementing standard relational database functionality with advanced transaction management and exception handling to support process support environments like workflow management systems [3]. This aim has been brought into practice by reengineering the existing FORO 'basic' workflow management system combined with an indexed file system. Figure 1a shows the basic FORO system, essentially consisting of a workflow engine, shielded from the file system by a software library forming an access layer, and a workflow client. In WIDE, this architecture is extended with transaction support (TS) and rule support (RS), and the file system is replaced by a relational DBMS, as depicted in Figure 1b [4]. For reasons of brevity and clarity, we focus on the design of subsystems of the TS extension in this paper.

To extend the basic FORO system, a classical reengineering cycle has been followed, in which an essential model of the current system has been abstracted, then transformed into an essential model of the desired system, and finally mapped to and implemented in a physically distributed environment. Below, we present a rational reconstruction of the design process.

4.1 The Desired Essential Architecture

The essential architecture abstracts from the distribution of the software over physical resources (allocations of runtime processes to workstations or servers) and is motivated exclusively in terms of the desired functionality of the system. To obtain an overall essential architecture of Figure 1a, we ignore the fact that the access layer is a function library without 'own' process and model it as a separate module, arriving at the architecture in Figure 2a. To transform this architecture to the desired essential architecture, we make two steps. Firstly, we observe that workflow transactions are supported in WIDE by an orthogonal two-layer transaction model [7], consisting of a global transaction layer and a local transaction layer. Consequently, we decompose the transaction support extension accordingly into two separate subsystems: global transaction support (GTS) and local transaction support (LTS), where only LTS requires access to underlying DBMS transaction facilities. Secondly, we replace the file system by the DBMS. After these steps, we arrive at the essential architecture depicted in Figure 2b.

Next, we further decompose the GTS and LTS subsystems. To perform this decomposition, we observe that at run-time, the WF engine contains a number of active workflow process instances (workflow cases), the lifespan of which is coupled to starting and ending workflow processes.

The GTS subsystem manages global transactions that have a one-to-one relationship with active workflow instances. Global transactions are dynamically created at workflow start and disposed of at workflow completion. Their state is influenced by events in corresponding workflow instances and stored persistently through the access layer. Apart from state administration, logic is required to compute compensating global transactions in case of global abort events [7]. Consequently, the GTS subsystem is decomposed into a dynamic set of global transaction (GT) modules that interface with the WF engine (event signaling) and the access layer (persistent storage), and a global transaction manager module containing the compensation logic.

The LTS subsystem manages local transactions that have a many-to-one relationship with active workflow instances [7]. Local transactions are created dynamically at the start of certain tasks in a workflow and disposed of at the completion of these tasks. Their state is influenced by workflow events in corresponding workflow instances. Because of their dynamic nature, local transactions are mapped to local transaction (LT) modules. To relieve the WF engine from the one-to-many communication between workflow instances and local transactions, a local transaction manager (LTM) is inserted between engine and LTs to perform a dispatching function (corresponding to the many-to-one pattern in [6]). As LTs perform transactional operations on the underlying DBMS and abstraction is required with respect to specific DBMS platforms, a local transaction interface (LTI) is inserted between LTs and DBMS. For transactional operations, the LTI acts like the access layer for data manipulation operations.

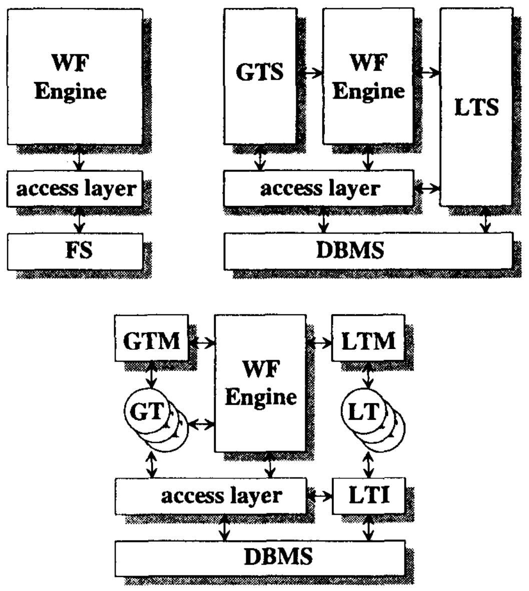

Figure 2 a, b, c: Essential architectures of basic system, desired system, detailed desired system

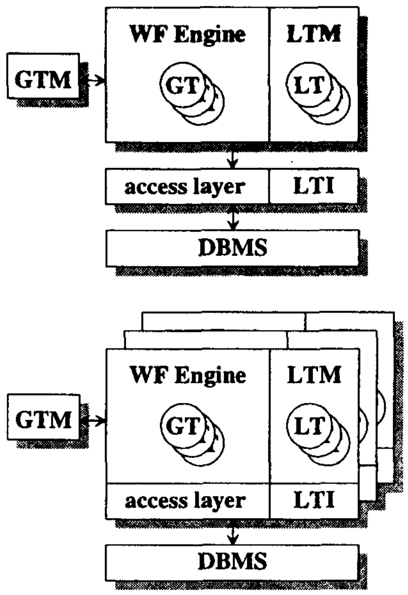

Figure 3 a, b:Intermediate physical architecture, final physical architecture

The result of GTS and LTS decomposition is shown in Figure 2c. This is the desired essential architecture of the extended WFMS. The next step is to map this essential architecture onto a physical architecture.

4.2 The Desired Physical Architecture

To go from essential to physical architecture, we have to take software clustering into processes and allocation of these processes into consideration.

We first observe that each GT and LT instance is related to one workflow instance, and that communication between GT/LT instances and workflow instances is frequent. Hence, it is efficient to place GT/LT instances in the same process. LTM functionality is relatively simple and communication between workflow instance and LTM is frequent, so it is efficient as well place the LTM in the same process as the WF instance. Functionality of the GTM, on the other hand, is relatively complex and communication between GTM and other modules is infrequent, so there is no reason to combine the GTM with other modules. This means that workflow engine, LTM and GT/LT modules are clustered into one process, but that GTM is kept as a separate process. As access layer and LTI perform similar and relatively simple functions, access layer and LTI are clustered. The result is shown in Figure 3a.

Next, clustering of extended access layer and extended workflow engine is considered. Observing that the state of both extended access layer and extended workflow engine can be partitioned on a per-workflow-instance basis, clustering is possible. Given very frequent communication between engine and access layer, clustering is desired, resulting in the physical clustering shown in Figure 3b.

The clusters can be replicated arbitrarily, using a flexible coupling of workflow engines to GTM servers. To allow for this flexibility in a transparent way, middleware functionality is required. In WIDE, a CORBA object request broker [9] has been used. This allows for flexible instantiaton at runtime with transparent allocation. To allow for flexible coupling of workflow engine to DBMS server, a client/server coupling has been used.

5. DISCUSSION AND CONCLUSION

The WIDE system has a number of characteristics relevant for the design of its architecture as discussed above:

- It is *distributed*, as dictated by the inherently distributed nature of workflow applications, requiring a well-structured design into distributed physical modules.
- It is a *general-purpose* system to be parameterized for specific workflow applications, requiring a high level of flexible approach to module replication and allocation.
- It is strongly related to highly complex business processes, requiring an *extensible* architecture with respect to support for advanced features like extended transaction management.
- It has ‘soft’ real-time character related to supporting business processes, requiring special care must be taken in the design of the communication structures such that *performance* is guaranteed.

We now discuss how these characteristics are reflected in guidelines used in the design process, conjecture that these guidelines are generally applicable to the design of extensible, general-purpose, distributed software systems.

We define an *essential decomposition* of a system as a decomposition that is motivated entirely by properties of the external environment of the system and not by the implementation environment [14]. Decomposition criteria for an essential decomposition may refer to external entities, the application domain or external functionality, but not to the distribution architecture of the underlying implementation platform or properties of the programming language. Like

[1], at this level of abstraction, we assume unlimited concurrency. This agrees with the classical concept of essential model defined in [8], which assumes perfect technology as an implementation platform. In the design of the WIDE system, *desired external functionality on the conceptual level* has been used as a criterion for decomposing the system at the essential subsystem level (for example, the conceptual transaction model). The reason is simple: for general-purpose systems, there is no specific application environment in terms of which we can partition the system, but there is a clear idea about the desired functionality.

The starting point for the software engineer in a systems engineering project is therefore a functionally decomposed system and it is good advice to consider decomposing the software *at the subsystem level* by the same principle. For an essential decomposition, other criteria that refer to the external environment should also be considered. For example, in control-intensive systems, one may define subsystems for particular classes of external devices or for particular classes of events, that share periodicity properties [13, 5, 16]. It is only when we decompose subsystems into finer-grained entities that the choice becomes relevant between encapsulating behavior and state into objects or separating processes from stores. It can be shown that this issue is distinct from the issue whether or not to use functional or application-domain-oriented decomposition [15].

Guidelines that merely list example kinds of subsystems, such as performance monitoring or alarm handling, should be treated with caution. What is needed is the reasoning behind identifying these examples, not the examples themselves. Using high-level functional concepts is essential in subsystem identification in general-purpose systems. In the WIDE design process, the observation that the concept of transaction is an essential ingredient to the architecture design, i.e., the development of a separate transaction support subsystem. The observation that transactions are conceptually split into two layers has resulted in splitting the essential architecture of the TS subsystem into two orthogonal modules (GTS and LTS). Basing an architecture on high-level functional concepts also contributes to the flexibility and extensibility of the system: changing or extending the system with modified or new functional concepts will easily translate to changes to the essential architecture.

By distinguishing the desired essential model from the desired (and realized) physical system, we create the freedom to map essential subsystem in various ways to physical resources, without changing the essential model. In the design process outlines above, we have shown how this mapping can be performed in a structured way to achieve desired distribution and performance characteristics. The essential model as the centerpiece of the design thus improves the traceability of the resulting software to essential subsystems and ultimately to requirements. This will highly contribute to extensibility and maintainability of the software. It must be noted, however, that a flexible implementation platform (like an object request broker infrastructure) for the physical architecture is crucial to the ability to use a clear separation between essential and physical architecture without sacrificing flexible distribution or performance.

6. ACKNOWLEDGEMENTS

All members of the WIDE team are acknowledged for their parts in the development described in this paper. Specific thanks go to Gabriel Sánchez, Stefano Ceri, Jochem Vonk, and Erik Boertjes.

7. REFERENCES

[1] M. Awad, J. Kuusela, J.Ziegler, *Object-Oriented Technology for Real-Time Systems: A Practical Approach Using OMT and Fusion*, Prentice-Hall, 1996.

[2] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, *A System of Patters – Pattern-Oriented Software Architecture,* Wiley, 1996.

[3] F. Casati et al.; *WIDE: Workflow Model and Architecture*; CTIT TR 96-19; University of Twente, 1996.

[4] S. Ceri, P. Grefen, G. Sánchez, "WIDE - A Distributed Architecture for Workflow Management". *Proc. 7th Int. Workshop on Research Issues in Data Engineering,* UK, 1997, IEEE.

[5] H. Gomaa, *Software Design Methods for Concurrent and Real-Time Systems*, Addison-Wesley, 1993.

[6] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns—Elements of reusable Object-Oriented Software,* Addison-Wesley, 1995.

[7] P. Grefen, J. Vonk, E. Boertjes, P. Apers, "Two-Layer Transaction Management for Workflow Management Applications", *Proc. 8th Int. Conf. on Database and Expert System Appl.*, France, Springer, 1997.

[8] S.M. McMenamin, J.F. Palmer, *Essential Systems Analysis,* Prentice-Hall, 1984.

[9] *The Common Object Request Broker: Architecture and Specification, V2.0*; Object Management Group, 1995.

[10] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, *Object-Oriented Modeling and Design,* Prentice-Hall, 1991.

[11] S. Shlaer, S.J. Mellor, *Object Lifecycles: Modeling the World in States*, Prentice-Hall, 1992.

[12] K. Shumate, "Structured analysis and object-oriented design are compatible", *Ada Letters*, 11(4), 1991.

[13] K. Shumate, M. Keller, *Software Specification and Design: A Disciplined Approach for Real-Time Systems,* Wiley, 1992.

[14] R.J. Wieringa, "Postmodern software design with NYAM: Not Yet Another Method", *Proc. NATO Worksh. on Requirements Targeting Software and Systems,* to be published, Springer.

[15] R.J. Wieringa, "A survey of structured and object-oriented software specification methods and techniques", to be published, *ACM Computing Surveys.*

[16] Yourdon Inc, *The Yourdon Systems Method: Model-Driven Systems Development*, Prentice-Hall, 1993.