Available online at www.sciencedirect.com
ScienceDirect
The Journal of Systems and Software 81 (2008) 727–746
The Journal of Systems and Software
www.elsevier.com/locate/jss

Integrating a software architecture-centric method into object-oriented analysis and design

Raghvinder Sangwan [a,*], Colin Neill [a], Matthew Bass [b], Zakaria El Houda [b]

[a] Pennsylvania State University, Great Valley School of Graduate Professional Studies, 30 East Swedesford Road, Malvern, PA 19355, USA
[b] Siemens Corporate Research, 750 College Road East, Princeton, NJ 08540, USA

Received 27 November 2006; received in revised form 23 July 2007; accepted 25 July 2007
Available online 10 August 2007

Abstract

The choice of methodology for the development of the architecture for software systems has a direct effect on the suitability of that architecture. If the development process is driven by the user's functional requirements, we would expect the architecture to appropriately reflect those requirements. We would also expect other aspects not captured in the functional specification to be absent from the architecture. The same phenomenon is true in development approaches that stress the importance of systemic quality attributes or other non-functional requirements; those requirements are prominent in the resulting architecture, while other requirement types not stressed by the approach are absent. In other words, the final architecture reflects the focus of the development approach. An ideal approach, therefore, is one that incorporates all goals, expectations, and requirements: both business and technical. To accomplish this we have incorporated, into a single architectural development process, generalized Object-Oriented Analysis and Design (OOAD) methodologies with the software architecture-centric method, the Quality Attribute Workshop (QAW) and Attribute Driven Design (ADD). OOAD, while relatively intuitive, focuses heavily on functional requirements and has the benefit of semantic closeness to the problem domain making it an intuitive process with comprehendible results. Architecture-centric approaches, on the other hand, provide explicit and methodical guidance to an architect in creating systems with desirable qualities and goals. They provide minimal guidance in determining fine-grained architecture, however. The integrated approach described in this paper maximizes the benefits of the respective processes while eliminating their flaws and was applied in a eight university, global development research project with great success. A case study from that experiment is included here to demonstrate the method.

Keywords: Software architecture-centric methods; Object-Oriented Analysis and Design (OOAD); Quality Attribute Workshop (QAW); Attribute Driven Design (ADD)

1. Introduction

Software architecture is defined as "... the structure or structures of the system, which comprises software elements, the externally visible properties of those elements, and the relationships among them (Bass et al., 2003)." Given this definition, every software system has an architecture; and, if one were to implement a system two different ways, each would have its own architecture. Which of the two, one might ask, is the preferred architecture? After all, both versions of the system support the same functionality. What makes one superior to the other? The answer depends on the context. If system functionality is the only important consideration and non-function requirements such as performance, maintainability and extensibility are not, for example, the clear choice may be the architecture that is cheapest to build. If, however, the consideration is an architecture that supports the creation of a product line, those non-functional requirements become critical and it may be the case that neither of the two architectures are suitable. Therefore, the suitability of an architecture is measured in terms of its fitness to purpose.

* Corresponding author. Tel.: +1 610 7255354; fax: +1 610 6483377.
E-mail address: rsangwan@psu.edu (R. Sangwan).

0164-1212/$ - see front matter © 2007 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2007.07.031

Heretofore fitness to purpose has been a technical consideration where the system's purpose is defined by the requirements specification. Considering that use cases (Cockburn, 2000) are the most popular form of requirements specification in use today (Neill and Laplante, 2003) and that it is only recently that effective approaches to documenting non-functional requirements in use cases have been proposed (Alexander, 2003; Zou and Pavlovski, 2006) it is not surprising that when the system's fitness to purpose is related to systemic, non-functional requirements, such as memory and temporal performance, architectures designed to maximize functional cohesion can fail. This situation is further exacerbated when the individual components of a system are developed independently of one another, as has become the trend in globally distributed system development. Whereas in local development important systemic properties not explicitly identified in the architecture might still be considered by the development team, since they all have a reasonable perspective over the entire system, in distributed development scenarios this overarching perspective is not shared and thus it is easy for systemic constraints to be overlooked resulting in blown budgets for memory or temporal performance, for example (Mullick et al., 2006).

To mitigate this risk we can adopt alternative approaches to architectural design where systemic properties, or quality attributes as they are called, are a primary driver. Of course, in commercial software development the real purpose of a system is described by overarching business goals, a concept that Hohmann (2003) described as the marketecture of a system – the business perspective of a system's architecture that directly influences the technical architecture. Given this consideration it is critical that these business goals actually drive the architectural design. One approach that has shown success in this area is the combination of the Quality Attribute Workshops (Bachmann et al., 2002; Barbacci et al., 2000) and Attribute Driven Design (Bass et al., 2003). As will be demonstrated later, this approach does indeed guide the architect in creating an architecture that reflects the business goals of the system under development but with the drawback that the fine-grained design is undefined. Alternatively, the standard approaches to object-oriented analysis and design, most comprehensively expressed by Cheesman and Daniels (2001), result in fully detailed fine-grained architectures with a high degree of semantic closeness to the problem domain that make them intuitive and easily understood. There is nothing in these approaches, however, that explicitly addresses the non-functional and systemic properties of the system and so we find that these aspects are not prominent in the resulting architecture. We have, then, two alternative paradigms for architectural design: architecture centric and OOAD. Each paradigm has significant advantages, but corresponding weaknesses.

We propose in this paper an approach that combines two examples of these alternatives to capture the best of both "breeds" in a single architectural development process. In brief, this process starts by capturing the business and technical goals of the system under design with QAW, then iteratively elaborates the coarse-grained architecture from a monolithic starting point through attribute driven design before applying standard OOAD techniques to determine the fine-grained architectural detail for the coarse grains generated by ADD. This integrated approach was applied in an ongoing worldwide software development project involving eight universities across four continents, and we present a case study example from this project that demonstrates the efficacy of the hybrid process.

In the proceeding section the Global Studio Project will be introduced describing a software development project used as a test bed for this investigation. Section 3 uses the OOAD approach described in Cheesman and Daniels (2001) to develop the architecture for the system under consideration in this project. Section 4 uses architecture-centric methods developed at Software Engineering Institute (SEI) such as Quality Attribute Workshop (QAW) and Attribute Driven Design (ADD) for the same purpose. Section 5 proposes an integrated approach that combines activities from OOAD with those from architecture-centric methods to create architectures that adequately support the business goals. Section 6 discusses related work, and Section 7 follows with conclusions.

2. Case study: global studio project

Siemens Corporate Research (SCR), in collaboration with eight universities, across four continents is currently conducting a multi-year experiment, called Global Studio Project (GSP), to gain a better understanding of the issues surrounding and the impact of various practices in managing globally distributed software development projects. At the time of this writing, the universities (shown in Table 1)

Table 1
Participating universities

University	Country	Dates involved		Role
		Start	End	
Pontifical Catholic University	Brazil	08/2005	07/2006	Student teams
Technical University of Munich	Germany	11/2004	06/2006	Student teams
International Institute of Information Technology Bangalore	India	11/2004	06/2006	Student teams
University of Limerick	Ireland	11/2004	06/2006	Student teams
Carnegie Mellon University	USA	09/2004	03/2006	Student teams
Harvard Business School	USA	07/2004	Present	Advisory
Monmouth University	USA	09/2004	04/2006	Student teams
Penn State University	USA	07/2004	Present	Advisory

contributing student teams to the development effort had completed two years of their involvement in this study.

As a part of GSP, student teams from these universities are to collaboratively develop a unified management station (called MSLite) for the building automation domain that will automatically monitor and/or control the internal functions of buildings, such as heating, ventilation, air conditioning, lighting, access and safety. The intended users of MSLite are facility managers who need to operate the many (hardware) systems required to support building functions. Since there are a large number of these systems, a Field System Simulator (FSS) is used during software product development to simulate these systems. An FSS configuration file is used to create the initial configurations of the simulated systems, including their structure and the initial values of their various properties. For example, a system that monitors air conditioning on some floor of a building may have several temperature sensors at various locations, and their threshold values for how to regulate the temperature may be set to the desired levels within the FSS file. Fig. 1 illustrates this broad functional context of the MSLite system.

Some of the high level functional requirements for the MSLite system are:

- Manage the field systems represented in FSS
- Issue commands to the field systems to change values of their properties
- Define rules based on property values of field systems that trigger reactions and issue commands to field systems
- Define alarm conditions similar to rules that when met trigger alarms notifying appropriate users

If SCR was to commercialize the MSLite system, it would need to do so by entering new and emerging geographic markets and opening sales channel in the form of Value Added Resellers (VARs). VARs sell the software under their own brand to support hardware devices of many different manufacturers. It is clear that these business goals would have a significant effect on the architecture of the MSLite system without necessarily affecting its functionality. For example, hardware devices from many different manufacturers would need to be supported and considerations would have to be made to take language, culture and regulations of different markets into account. Trade-offs would need to be made and risks assessed to determine the extent to which the product should support these goals. Depending on the company's comfort level with the trade-offs and risks, these goals may need to be refined, e.g. scaling back on the intended markets.

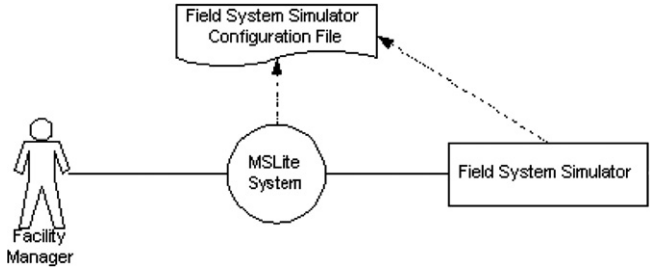

Fig. 1. Broad functional context of the MSLite system.

All of these business decisions require input from technical staff to determine the impact of such requirements and to inform the technical staff of the importance of these systemic requirements. Too often, however, there is a disconnect between what an organization wants and what its technical team delivers. We have come across a case where a business unit wanted to create a high performing infotainment system for a luxury line of cars in a compressed time to market. The technical team was forced to distribute the development of parts of this system across geographically distributed teams to achieve the compressed schedule via parallel development efforts. When the components developed by the teams were integrated together, they blew up the memory and performance budget. While individual components were carefully crafted, not enough attention had been given to the overall system goal of achieving high performance within the given resource constraints. End result was the business unit was not able to produce the desired product. The disconnect between what was desired and what was delivered cost the company hundreds of millions of dollars spent developing the system and billions of dollars in potential revenue. Clearly, there is a need to bridge this gap.

To avoid a similar situation, the teams developing MSLite would, therefore, need to pay special attention to the business goals of entering new and emerging geographic markets with the accompanying demands on modifiability and interoperation when opening new sales channels in the form of VARs. Given these business goals for the system we will now consider the two approaches to system design starting with the typical and familiar approach to object-oriented development where the software design is based on the model of the problem domain.

3. The OOAD approach

In order to create the architecture of a system, the OOAD first captures the requirements by identifying the user–system interaction at the boundary of the system under consideration. These interactions are described in the form of use cases. One of the ways of identifying use cases is to begin analyzing the business processes a system will be designed to support (Cheesman and Daniels, 2001). A business process is a sequence of steps or activities the business workers undertake to provide a service or perform a task. While the system under design could potentially automate the entire business process, typically activities within the business process that are candidates for automation have to be identified.

Activities within a business process result from business events initiated by people or external systems, referred to in the Unified Modeling Language (UML) as actors. The

activities triggered by a single event form a single use case and the actor responsible for the event becomes the primary initiator of the use case. Use cases are goal oriented; in other words, at the end of a use case the actor initiating the given use case walks away from the system with something of value (Cockburn, 2000).

A business process can be described using UML activity diagrams. Fig. 2 shows a subset of business processes the MSLite system must support.

These diagrams address a portion of the initial building automation domain. This was done to keep the illustrative analysis within a reasonable size for the objectives of this paper. Activity diagram in Fig. 2a depicts the building configuration workflow. This workflow is triggered when a facility manager receives a new building operation policy or an update to an existing policy. Depending on the policy details, a new rule may be created, an existing rule may be amended or no change may be required if the policy is already implemented by the existing rules. The rules can have different reactions such as issuing commands, generating alarms or sending notifications. In the activity diagram in Fig. 2b, the handling of alarms by a facility manager is shown. Again, depending on the alarm details, the facility manager may have to acknowledge an alarm, follow a Standard Operating Procedure (SOP) in response to the alarm or simply dismiss the alarm.

UML use case diagrams can be used for depicting the use cases for a system. Fig. 3 shows a subset of use cases and actors identified from the analysis of some of the business process to be supported by MSLite including the ones described by the activity diagrams in Fig. 2. These use cases are grouped into four use case packages. Use Case Package 100 (UCP100) is concerned with configuring the building operations and contains use cases for defining Automation rules, Alarms and Standard Operating Procedures. UCP200 covers some aspects of monitoring of the building health and includes use cases for Issuing commands to field devices and Handling Alarms and their lifecycle. UCP300 addresses the Personalization of the system by its operators (facility managers). Finally, UCP400 manages the interaction with field systems. It contains use cases for handling events originating from field systems which include for example changes of some field system property value and failure reports.

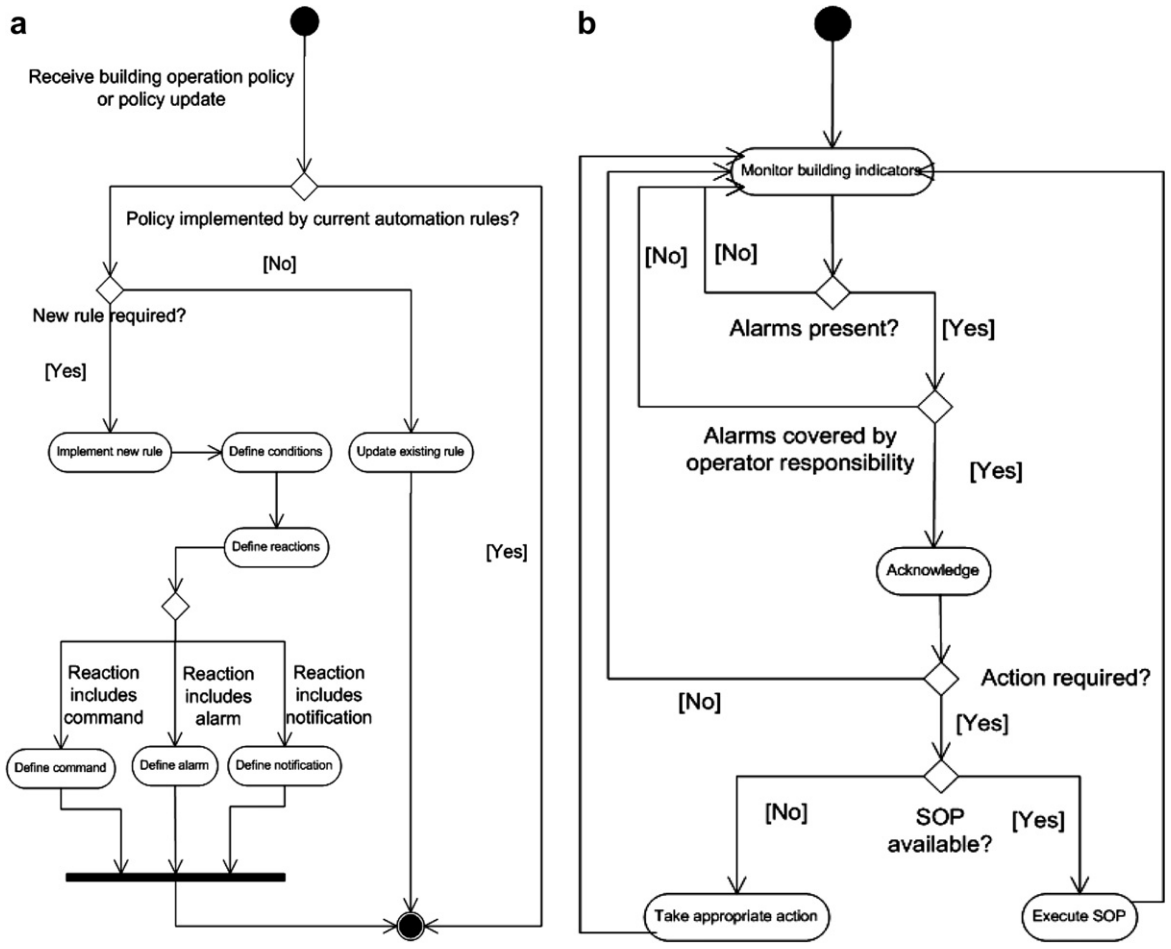

Fig. 2. Activity diagrams showing business processes; (a) shows activities related to configuring building operations and (b) shows activities related to monitoring the health of the building.

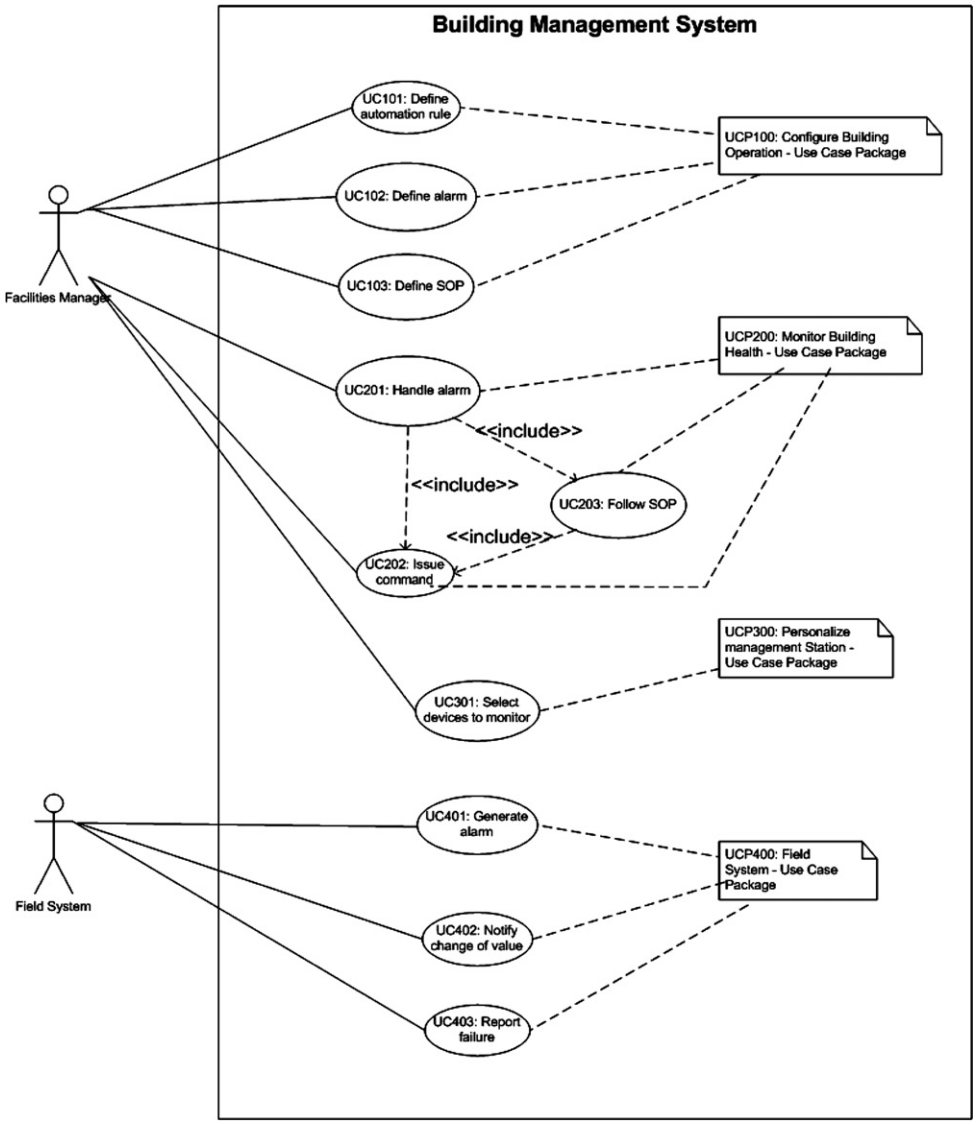

Fig. 3. Use cases for MSLite system.

The business process descriptions in Fig. 2 introduce some significant concepts from the building automation domain such as alarms, rules, commands, SOP, etc. These concepts are significant because they represent the business information/entities that get created, destroyed, associated and used in various ways within a use case in order to achieve something of value. Therefore, an additional important artifact in OOAD is a business concept model that captures these significant business terms and the relationships among them (Larman, 2004). Fig. 4 shows this model for the MSLite system using the UML class diagram.

The business concept model of the MSLite system identifies a collection of domain entities and qualifies the way they interact or are related to each other using associations. It is, for example, possible to see that Field Systems contain field devices of different types and are themselves linked by a network. A field device (such as a humidity detector) is located at a physical location and contains properties of various types which "read" the environmental conditions at that location.

The use case model and the business concepts model serve as inputs for specifying the components for a system that become a basis for its architecture (Jacobson et al., 1999). In order to specify the components, we must first identify the interfaces they support. The use cases being at the boundary of a system help identify the system interfaces. The business concepts representing entities utilized by the use cases help identify business interfaces used for managing these entities (Larman, 2004). Fig. 5 shows the system interfaces and business interfaces for the MSLite system.

In Fig. 5a system interfaces and their corresponding use cases are shown. Initially, methods for these interfaces are extracted from the use case steps. Fig. 5b shows the business interfaces. The process for obtaining these interfaces

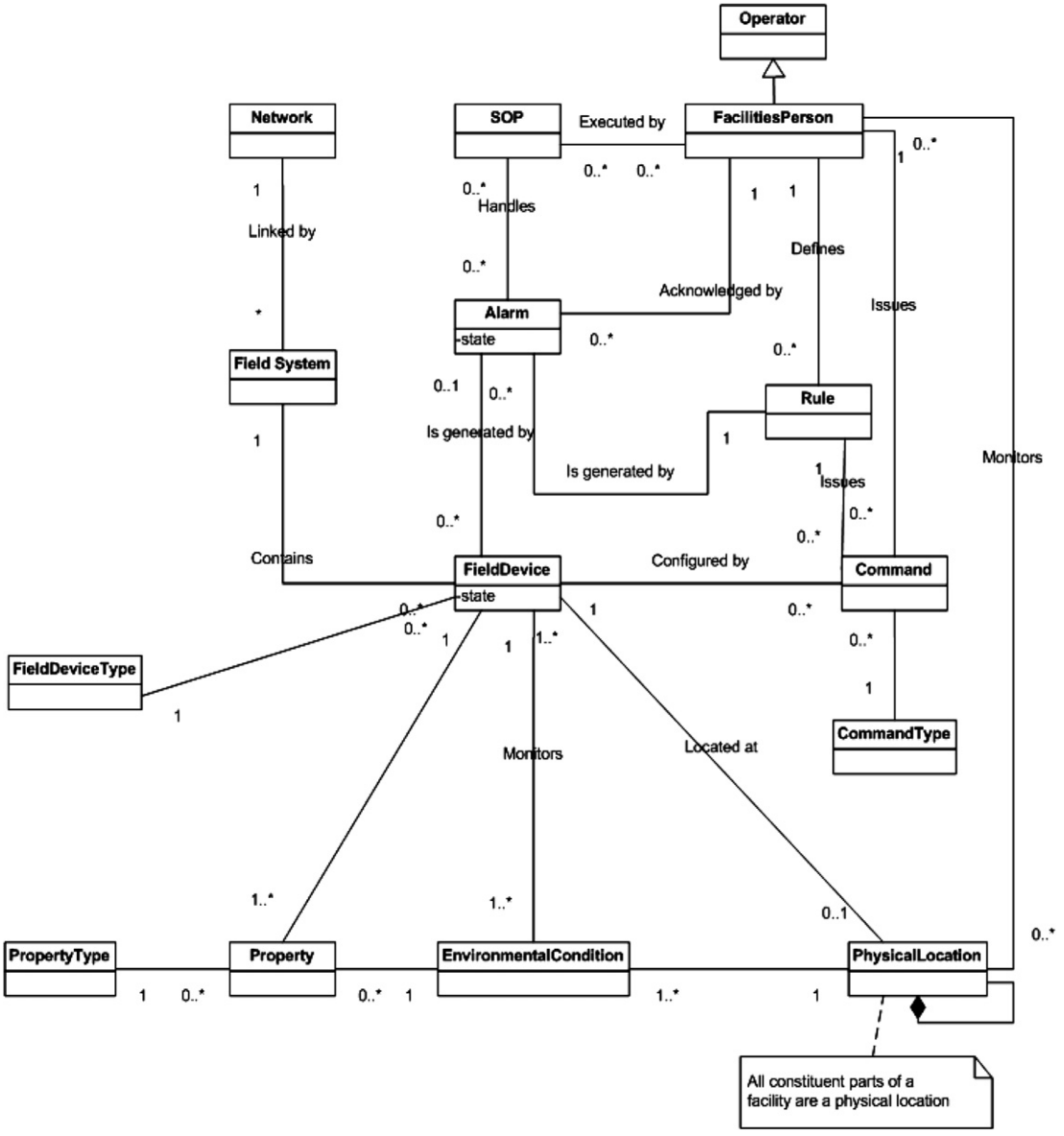

Fig. 4. Business concepts model for MSLite system.

starts by refining the business concepts into core business types. We identify core business types as business concepts in the Business Concept Model (see Fig. 4) that have no mandatory associations.[1] A business interface is then created for each core type. The resulting model shown in Fig. 5b indicates the core types using the ''core'' stereotype.

Once the interfaces have been identified, each interface (system or business) could be allocated to a single component or a single component could support multiple interfaces. This is where OOAD does not provide firm guidelines. The allocation of interfaces is primarily driven by the principles of abstraction, encapsulation, and separation of concerns that result in loosely coupled and highly cohesive components. While cohesion comes in many forms, the dominant form for most developers is that of functional closeness of the class members (Schach, 2006). One, therefore, is mainly performing functional decomposition of the system at this point with very little focus on its business goals or other quality attributes. Fig. 6 shows the system and business components for MSLite.

System components shown in Fig. 6a were obtained by regrouping interfaces dealing with the same functional aspects of the system. The diagram shows business interface dependencies to the left of each component. In Fig. 6b, a business component is created per business interface.

Once the initial component specifications, their supported interfaces and their interface dependencies have been identified, a component specification architecture for a system can be created such as the one shown for the

[1] The exception is made for associations with categorizing types. Refer to Cheesman and Daniels, 2001 chapter 5 for a more detailed description of the process applied.

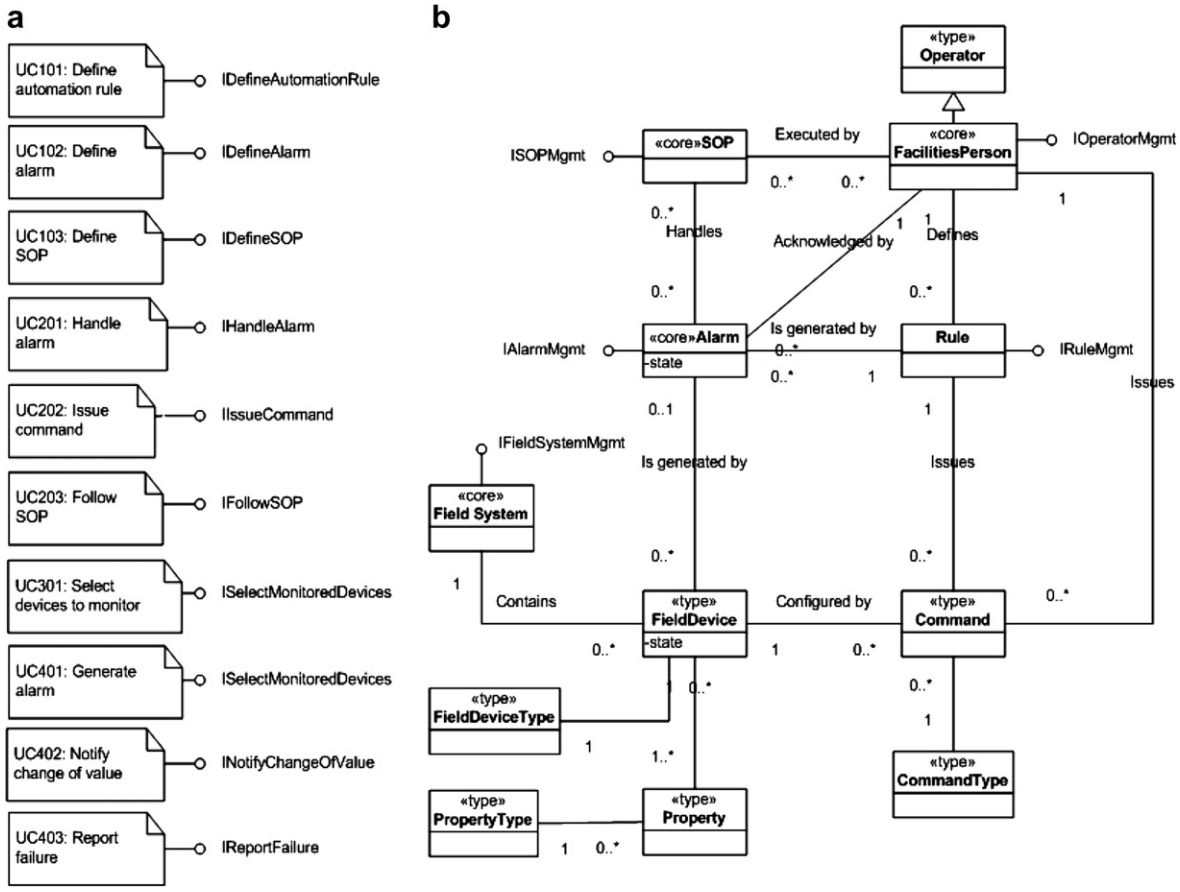

Fig. 5. (a) System interfaces identified from use case model, and (b) business interfaces identified from business concepts model.

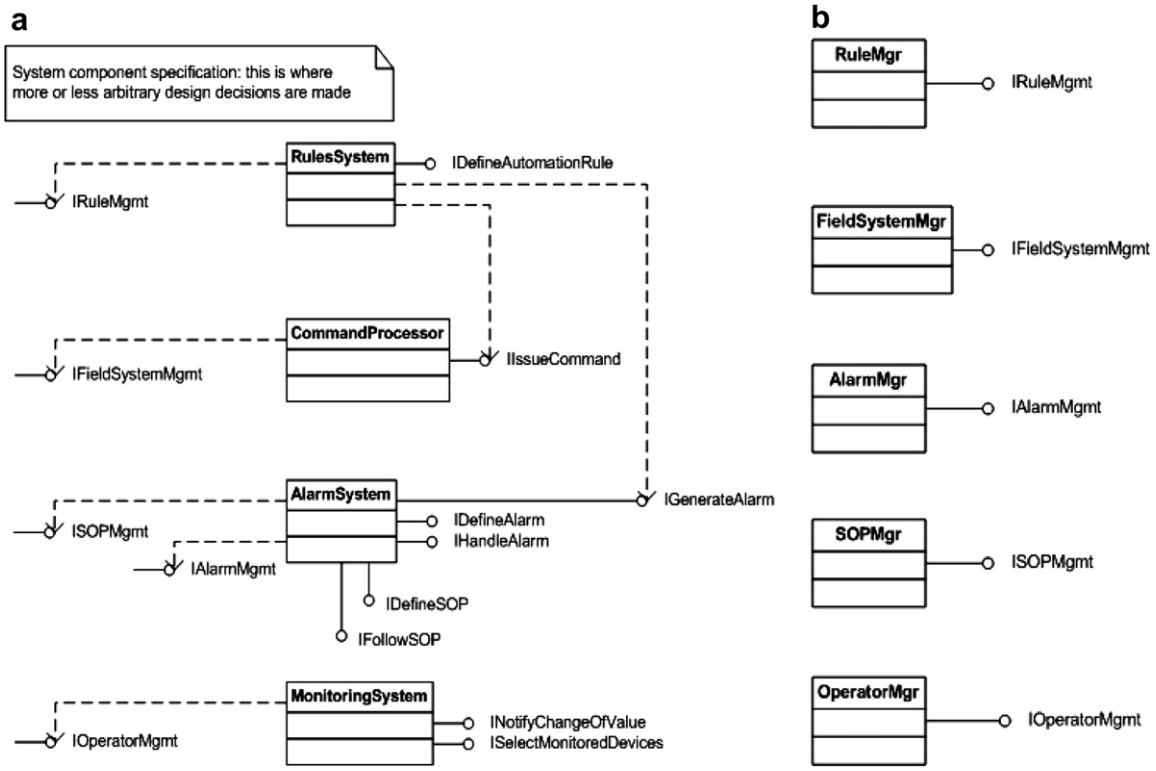

Fig. 6. (a) System components, and (b) business components for MSLite system.

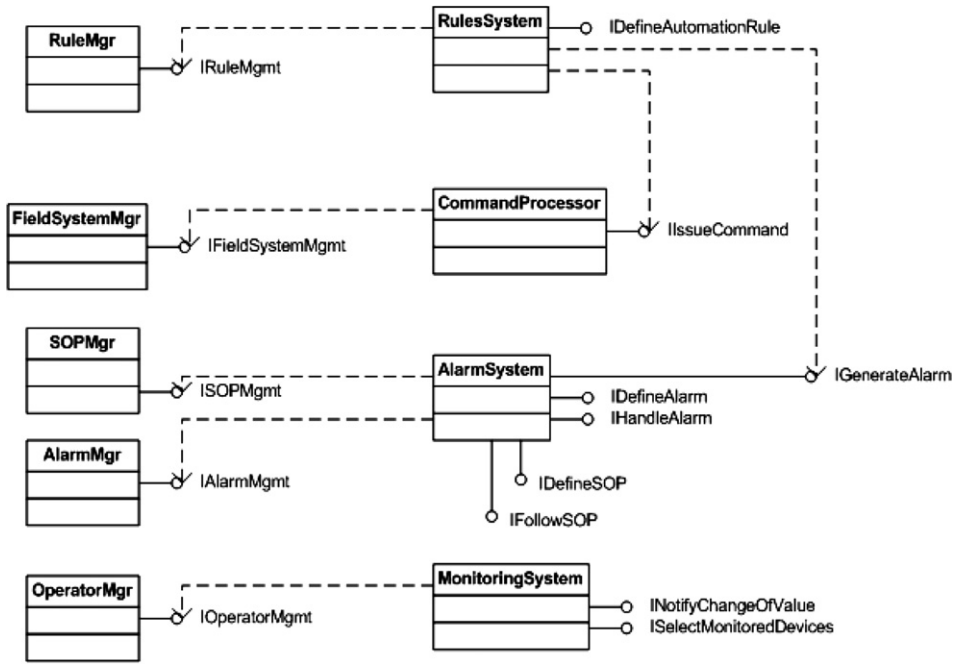

Fig. 7. Components specification architecture for MSLite system.

MSLite System in Fig. 7. This results from combining system components and business components from Fig. 6.

It should be noted that we have limited our analysis and design to a small fraction of the MSLite system. In reality there are many more business and system components than those shown in Fig. 7.

The architecture obtained in Fig. 7 reflects the use case driven nature of OOAD. The system components were created by aggregating functionally cohesive use cases. It also shows OOAD's closeness to the model of the problem domain. The business components were motivated by identifying core entities in the business concepts model. Although this approach does lead to architectures with loosely coupled and highly cohesive components that are easy to understand due to their semantic closeness to the problem domain, it is predominantly using functional decomposition with very little focus on the business goals or quality attributes. In the given example, for instance, the quality attribute requirements associated with supporting new and emerging geographic markets, and different VARs are not explicitly accommodated in this architecture. These requirements call for modifiability concerns such as adding a new hardware device or supporting a new language but without further refinement and the introduction of adaptors and factories such changes to the architecture later in the development lifecycle would be costly. Moreover, the development approach itself does not highlight such non-functional requirements so possible accommodations are not usually even considered.

4. The architecture-centric approach

In contrast to the OOAD approach of the previous section, architecture-centric approaches focus on systemic properties that the software architecture must embody. Factors that influence the architecture, therefore, tend to be the quality attributes such as performance, modifiability, security and reliability (Bass et al., 2003). Consequently, these quality attribute requirements become a starting point for the architecture-centric methods. Of course, these requirements must provide sufficient detail in order to be truly useful. For instance, it may not be sufficient to say that a system must be modifiable. Any system is modifiable with respect to something, and a system can be modified with respect to any aspect given enough time and money. The question is modifiable with respect to what, when and with how much effort?

Since they are the drivers for the architectural decisions, the first task is to determine the important systemic properties. This is done with Quality Attribute Workshops (Bachmann et al., 2002; Barbacci et al., 2000) – an architecture-centric method for eliciting quality attribute requirements from the stakeholders of a given system. The goal of this method is to establish a prioritized set of architecturally significant requirements in the form of quality attribute scenarios that are mapped to the business goals. Clearly, it is important these goals are known before the workshop can be conducted even if they are initially very general and will need subsequent refinement. The MSLite system has the following business goals:

> BG1: In order to succeed in the Value Added Resellers market, the system must be able to support hardware devices from different manufacturers. This includes existing and to some extent future devices.
> BG2: It must be possible to modify the system to support different languages, cultures and regulations. As the first step, these goals can be further refined as follows:

BG2.1: The system must allow changing all user interactions language to a language of choice. This includes languages with non-Latin characters and scripts written from right to left.
BG2.2: The field devices supported by the system can use different units. These units can be different from the units used by the user when specifying automation rules thresholds and commands. The system must be able to make all required conversions for rule evaluation and commands without errors and without user intervention.
BG2.3: Certain regulations and certifications require all life critical systems such as fire alarms and intrusion detection systems to operate within specific latency constraints. The system must be able to meet these latency requirements with a sufficient margin.

The next step is to link these business goals to the corresponding quality attributes as shown in Table 2. The table also shows tactics (Bass et al., 2003) that can be used for addressing quality attribute requirements when elaborating the architecture for the MSLite system.

Finally, for each quality attribute, the scenarios characterizing the corresponding quality attribute requirements are summarized in Table 3. This table also shows a priority evaluation for each scenario. The first value represents the importance of the scenario to the stakeholders. The second is an evaluation by the architecture team of the difficulty to implement that scenario. Scenarios that are a high priority (H) to the stakeholders and have a high (H) degree of difficulty in implementation will be addressed before those with low priority (L) and low (L) degree of difficulty. M represents medium priority and medium degree of difficulty.

Table 2
Business goals, corresponding quality attributes and tactics

Business goal	Quality attribute	Tactics and tactic categories
BG 1	Modifiability	*Localize change*
BG 2.1	Modifiability	– Anticipate expected changes
BG 2.2	Modifiability	– Generalize module
		Prevention of ripple effect
		– Use an intermediary
		– Maintain existing interfaces
		– Hide information
		Defer binding time
		– Runtime registration
BG 2.3	Performance	*Resource demand*
		– Increase computational efficiency
		– Reduce computational overhead
		Resource management
		– Introduce concurrency
		Maintain multiple copies

After the architecturally significant requirements have been elicited, the architecture that meets these requirements is elaborated using attribute driven design (ADD) approach (Bass et al., 2003).

4.1. Architecture elaboration

The architecture elaboration approach we use as part of ADD is an iterative process. ADD starts by treating a system as a single monolithic component responsible for all of

Table 3
Quality attribute characterization

Quality attribute	Attribute characterization	Attribute scenarios	Priority
Modifiability/ extensibility	Support for new field device system	E1. Support for a new Field Device System offering functionality comparable to the field system simulator must be added. The configuration information and details of the interface (calling conventions, method names, etc.) are in a different format. A team of two developers reasonably experienced with C# extends MSLite to support the new system in 320 person hours (40 h per week and person, 4 weeks)	(H, H)
	International Language Support	E2. A new language needs to be supported by the system. No code modification is required. A developer reasonably familiar with the system is able to package a version of the system with the new language in 80 person hours (40 h per week and person, 2 weeks) excluding string translation time	(H, M)
	Non-standard units support	E3. A new field device system using non-SI units is connected to the system. A system administrator configures the system to handle the new units in less than 3 h	(H, M)
Performance	Latency of event propagation	P1. A field system detects a change of a property value and notifies MSLite. The system operates under normal conditions.[a] The value is updated on all user screens that currently display the property value within 3 s. The time durations specified in this scenario are performance goals and not hard deadlines	(H, H)
	Latency of alarm propagation	P2. An event which should trigger an alarm is generated in a field device. The system operates under normal conditions.[a] The alarm is displayed on the user interfaces of all users that must receive the alarm within 3 s after the generation of the event	(H, H)

[a] Normal conditions are specified as follows:

- Number of concurrent sessions connected to MSLite < 15.
- Change of value (COV) rate < 600 per minute (30 field object properties at 20 COVs/min).
- Active automation rules $= 50$.

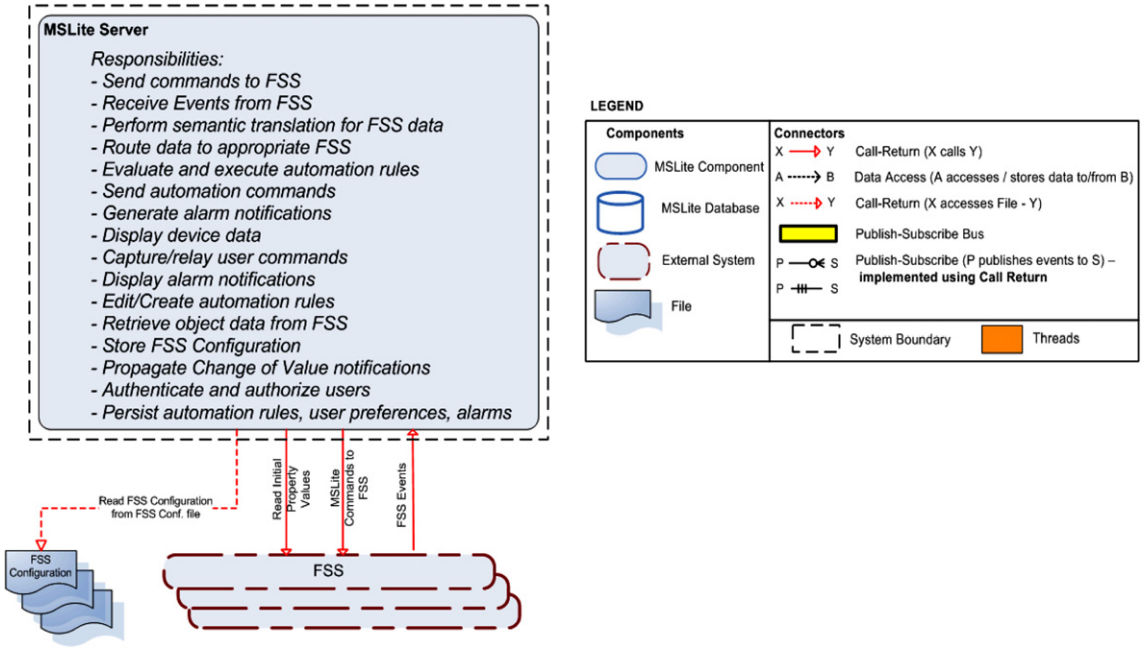

Fig. 8. A monolithic component.

the system functionality. It then recursively decomposes the system by applying architectural tactics successively to satisfy each quality attribute requirement. Very frequently, applying multiple tactics implies taking conflicting design decisions. This is why the end result of the architecture elaboration process is a compromise directly reflecting the quality attributes prioritization.

Initially, the system consists of a single component responsible for all the functionality to be implemented. This component is shown in Fig. 8 along with the common legend which will be used for all the component and connector diagrams produced during the elaboration process.

From the business goals linked to modifiability, we can observe that a primary variation dimension is the type and number of field device systems the MSLite system will have to interact with. In *anticipation of these changes*, the decomposition of the system attempts to minimize the number of components having a syntactic dependency on the field systems. This is achieved by introducing an adapter for each field system. The *anticipation of expected changes* tactic by itself has minimal benefit in reducing ripple effects when adding support for a new field system because it does not take into account indirect dependencies. We use two additional sub-tactics to minimize propagation of change. First we specify a standard interface to be exposed by all adapters (*maintain existing interfaces*). Additionally, we use the adapter as an *intermediary* responsible for semantic translation (when possible). This translation covers for example the unit conversions mentioned in business goal 2.2. Fig. 9 depicts the system after the introduction of the adapters.

Despite applying the tactics mentioned above, the MSLite server is still sensitive to a change in the number of field devices it is connected to, and must include logic

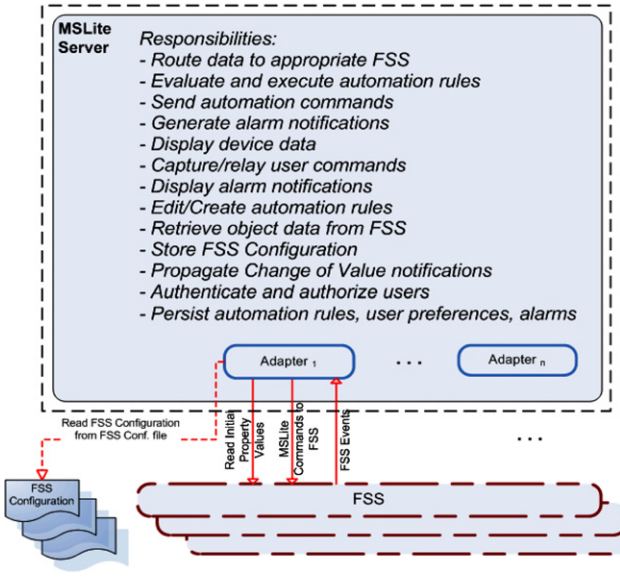

Fig. 9. Introduction of adapters.

to route commands and data to and from the correct adapter. *Hiding information,* another modifiability sub-tactic, is introduced to further limit the ripple effect when adding/removing field systems. This is done by introducing the concept of a Virtual Field System Simulator (VFSS). The VFSS hides information about the number and type of field systems actually connected. For all other components of the MSLite System, there is practically one field system to interact with at all times. The result of applying this tactic can be seen in Fig. 10.

At this point, we have applied most of the modifiability tactics identified in Table 1 which address field system var-

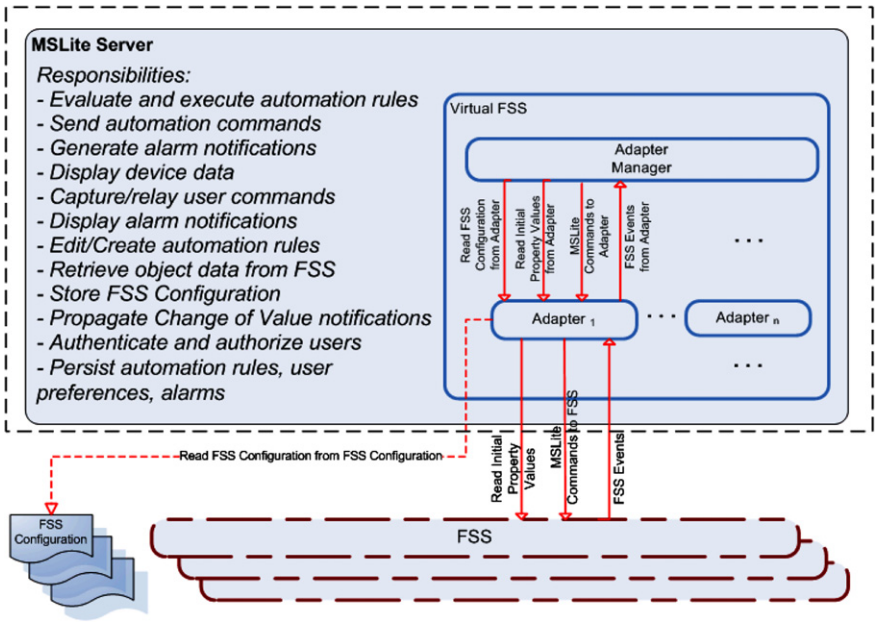

Fig. 10. Applying *hiding information* tactic.

iability. By doing this, we have actively included the ability to fulfill business goals 1 and 2.2 in the architecture elaboration. Other types of variability will be included further in the process.

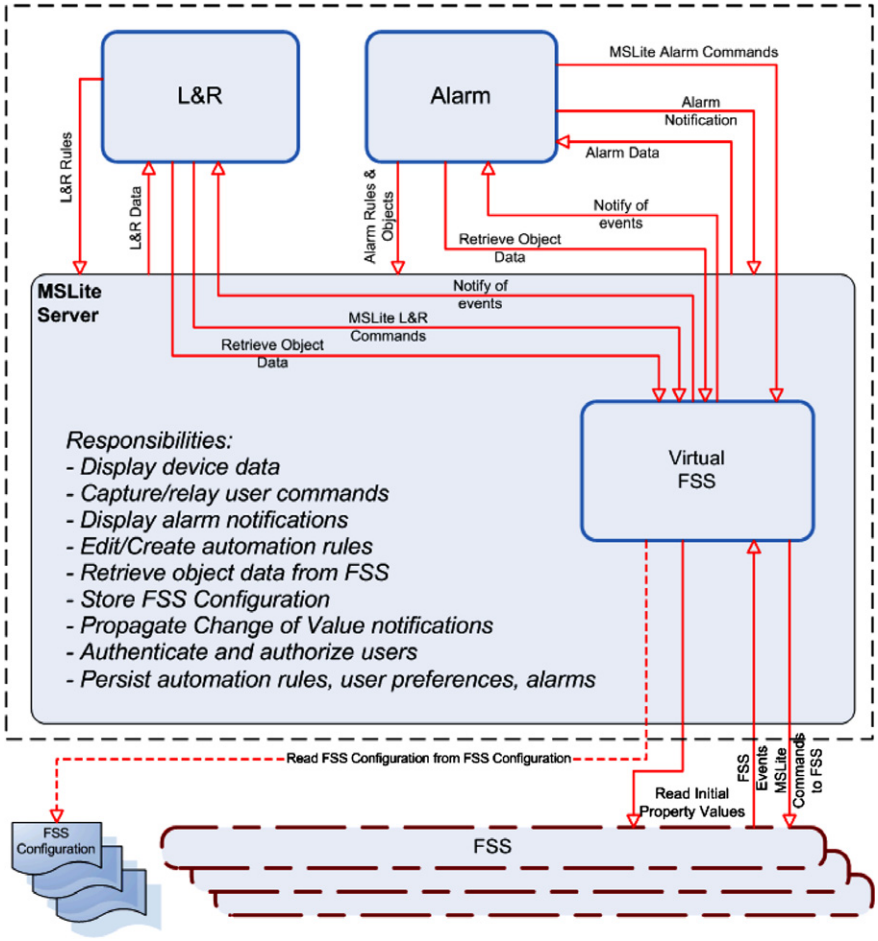

Fig. 11. Applying the *introducing concurrency* tactic.

In order to support business goal 2.3, performance tactics will be applied next. The first tactic belongs to the *resource management* category and relies on introducing *concurrency* to reduce delays attributable to "blocked time". The evaluation of automation rules is a prime candidate for concurrency since it is a computationally intensive process with a fairly low and predictable amount of communication. We therefore move the responsibility of rule evaluation and execution, and alarm generation, respectively to a separate Logic and Reaction (L&R) engine component and an Alarm engine component. These components running outside the MSLite Server context can be easily moved to dedicated execution nodes if necessary. Concurrency was also used inside these engines to perform simultaneous rule evaluations with the help of thread pools. This second application is however not visible at the level at which Fig. 11 shows the new structure of the system. It should be mentioned that the L&R and Alarm components may give the reader an impression that they are identical to the RulesSystem and AlarmSystem components in the architecture produced via OOAD. That is, however, not the case; the appearance of similarity comes from the way these components are named. We limited ourselves to the vocabulary of the domain when naming components rather than concocting artificial names.

The only business goal not currently incorporated in the architecture is business goal 2.1 which focuses on modifiability of the user interface. The modifiability tactic chosen to support this business goal is the *anticipation of expected changes* and their localization in a separate user interface presentation module. The separation of the user interface from the rest of the application is also classified as a usability tactic in Bass et al., 2003. It can be implemented using a variety of architectural patterns. In our case we chose a variant of the Model View Controller (MVC) pattern. The new state of the system's component and connector view can be seen in Fig. 12.

By examining the system structure in Fig. 12, it can be seen that every time the L&R, the Alarm or the Presentation component needs a value from a field device, it needs to make a call traversing multiple components all the way to the field systems. Since crossing component boundaries typically introduces computational overhead, and because the querying latency of field systems is a given constraint over which we have no control, we introduce a performance tactic relying on *maintaining multiple copies* of data to improve device querying performance. This is achieved by using the value cache component seen in Fig. 13. This cache provides field device property values to the other system components, saving part of the performance cost incurred when querying the actual field devices. The performance gains are seen because we reduce the number of process and machine boundaries traversed for each query.

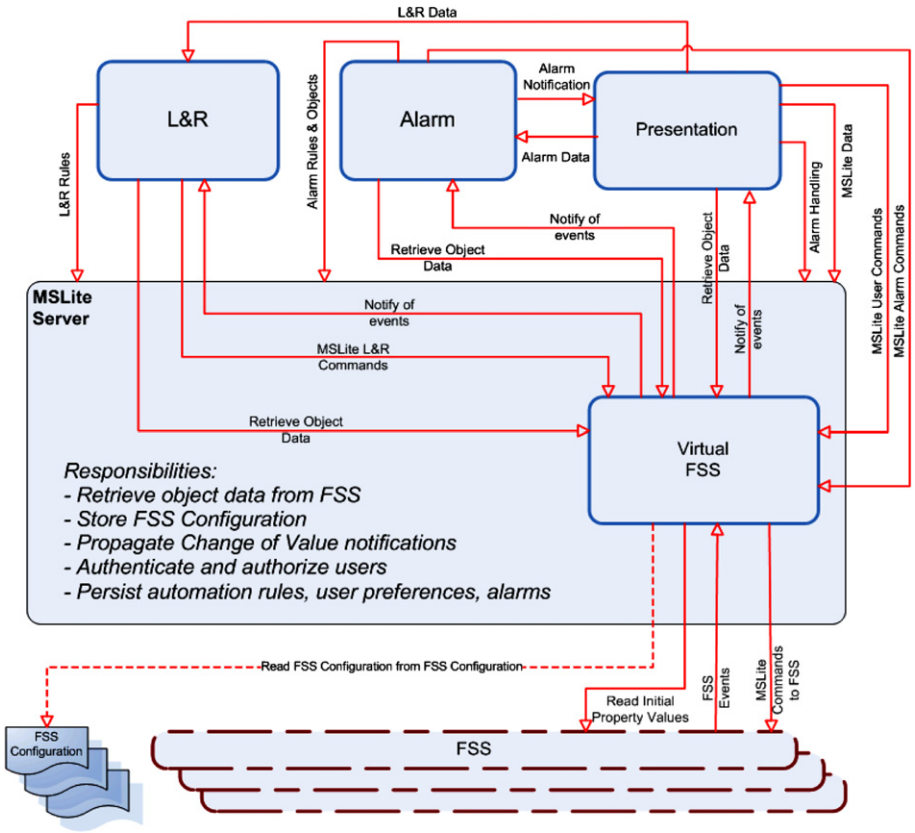

Fig. 12. Applying the *separation of user interface* tactic.

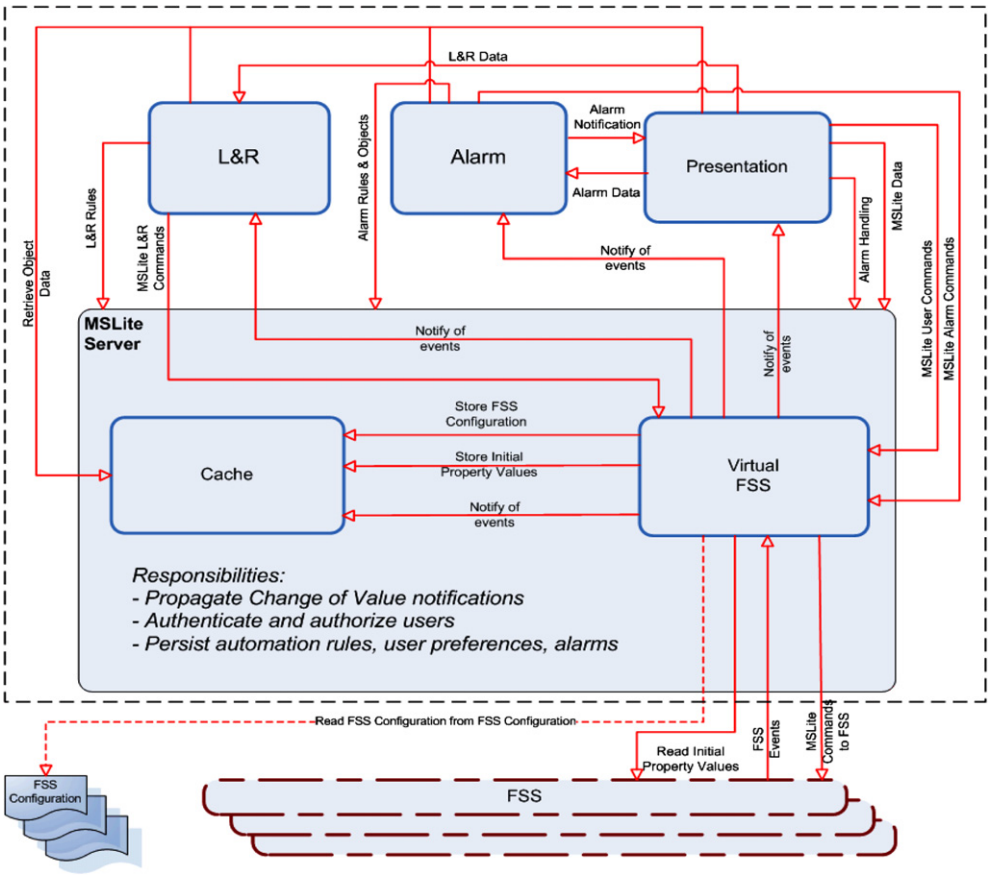

Fig. 13. Applying the maintaining multiple copies of data tactic.

The architecture obtained in Fig. 13 reflects the focus of the architecture-centric approach on systemic properties that the software architecture must embody. These systemic properties were used as a starting point for creating the architecture. Although this approach does lead to architectures that are more robust when a system's fitness to purpose is related to systemic or non-functional requirements, it does not address how subsequent design such as the process of identifying component interfaces and their respective operations must occur.

It should also be noted that in order to generate quality attribute scenarios for the architecture-centric approach some understanding of the overall functional requirements of the system is necessary. So activities similar to OOAD that establish a use case model and business concepts model (illustrated in Figs. 2–4) must also take place.

5. An integrated approach

In the previous sections we have explored the architectural analysis and design of the MSLite system and showed that using the ADD approach we arrive at an architecture that supports the business and mission goals of the application, but leaves the fine-grained design details unspecified. Correspondingly, the traditional OOAD approach arrives at a final design that includes these fine-grained, class-level, details, but these are distributed across an architecture that reflects an emphasis on functional cohesion rather than fundamental business goals. Ideally then we would prefer to merge the two approaches to arrive at a final architecture that simultaneously meets business goals and provides sufficient detail for implementation. Fig. 14 presents a process workflow for such an integrated approach that was used for developing the MSLite system. It should be noted that this is a partial view putting with emphasis on synergy points between the two methods. We voluntarily omit from the figure subsequent activities concerned with constraint specification, provisioning, etc. as they lie beyond the scope of the discussion.

Since quality attributes are central to creating an architecture, the architecture derived from architecture-centric methods should form the basis of design and implementation of the system under consideration. In the integrated approach the analysis and domain modeling activities in OOAD, such as the use case and business concepts modeling, that provide a broad understanding of the functional requirements, were performed concurrently and iteratively with activities in the architecture-centric approach that provide an understanding of the quality attribute requirements of the system. The quality attribute requirements were then used for further elaboration of the architecture. As architectures produced in this manner are high level

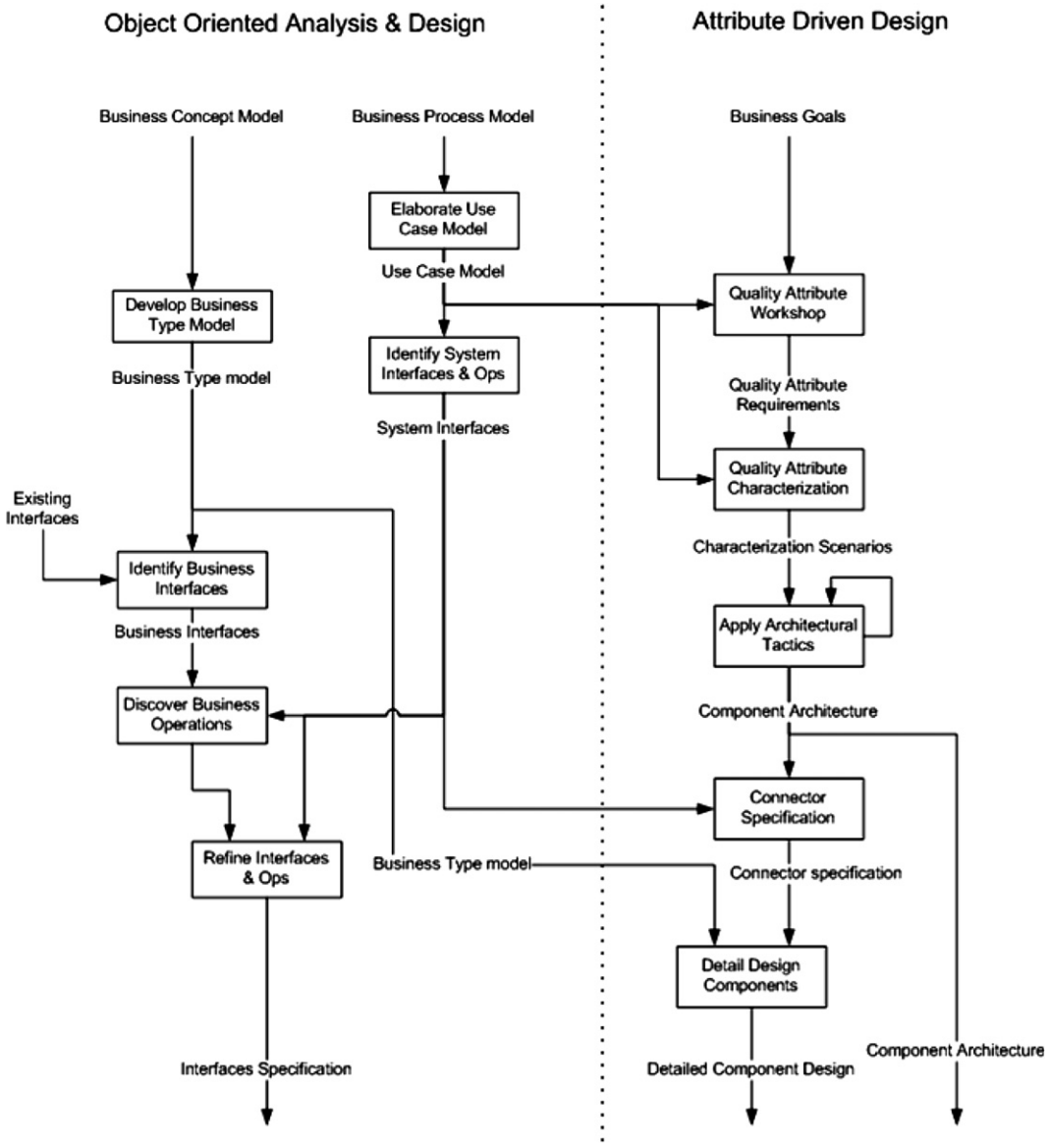

Fig. 14. Process workflow for an integrated approach.

models, detailed design and implementation of the components and their connectors was carried out using OOAD. Such a synergy between the OOAD and architecture-centric approaches provides linkage from high level models to the source code that is important for preserving the integrity of the architectural design as the system evolves (Garlan, 2000).

For the MSLite system, we carried out OOAD activities illustrated in Figs. 2–4 concurrently and iteratively with the architecture-centric activities illustrated in Tables 1 and 2. The OOAD activities took a broad and shallow approach in creating a fairly comprehensive list of the functionality to be supported by the system expressed in terms of the key concepts captured in the business concepts model. These became input for the quality attribute scenarios elaborated during the QAW. It is important to note that this broad and shallow approach implies enumerating most of the functional requirements but elaborating only the most important (for example, architecturally significant) ones. This is in essence an iterative and incremental approach as opposed to a waterfall approach.

With this understanding, the architecture was elaborated as shown in Figs. 8–13 using ADD. We started with the system as a black box and continued to decompose it using a prioritized list of quality attribute requirements obtained from the QAW. As particular decomposition of a system may conflict with a prior one and some trade-off may be necessary to manage the competing requirements. In case of the MSLite system, after focusing on modifiability, we introduced a number of performance tactics which resulted in the creation of multiple components as shown in Fig. 13. Based on the structure of these components and the type of their connectors, we predicted that some changes to the virtual FSS have the potential to prop-

agate to five other components (L&R, Alarm, Presentation, Cache and MSLite Server). This would be particularly damaging to the modifiability value of the system. As explained in Bass et al. (2003), the end architecture must strike the right balance in the classic Performance/Modifiability trade-off.

The Publish–Subscribe bus is a component we introduced to implement three modifiability tactics. First it alleviated the syntactic dependencies of inter-component calls by acting as a standard interface *intermediary*. Second, using the *module generalization* tactic it was made invariant to the type of events it transports. This generalization allowed new types of events to be transported with no modification to the Publish–Subscribe component. Finally, it relied on *runtime registration* to allow system extensibility by adding publishers and subscribers. The state of the system after the introduction of the Publish–Subscribe component is shown in Fig. 15.

The tactics we applied next were not explicitly stated in the subset of business goals mentioned earlier but are essential non-functional requirements. We briefly mention them here for completeness. Security requirements of the system were met by introducing user authentication in the access control module. "Buildability" was improved by delegating data persistence to an external commercially available database system. The final iteration of tactic application produced the system structure shown in Fig. 16. This decomposition became the basis for detailed design and implementation of the components and connectors shown.

It should be noted that the elaboration approach using ADD can also be applied recursively to the components in Fig. 16 for creating their respective internal architecture. We show this for the Logic and Reaction Engine in Fig. 17.

This runtime component and connector view of the Logic and Reaction Engine, shows the Rule Cache, Coordinator, Evaluator, PropertyMapper, Subscription Manager, Command Dispatcher and Event Queue components. When events are received, the Coordinator uses the Property Mapper to identify the rules to be evaluated and notifies an evaluator. The Evaluator retrieves the rule details from the Rules Cache and communicates the resulting commands to the Command Dispatcher. By using concurrency (thread pools) the component is able to achieve different types of performance gains. Some of these benefits however are mostly visible when multiple computational nodes are available. The corresponding static structure is depicted using a design class diagram in Fig. 18.

The class structure in Fig. 18 is shown in an intermediary state where main methods and attributes were identified and a reduced set of generalizations was applied. At

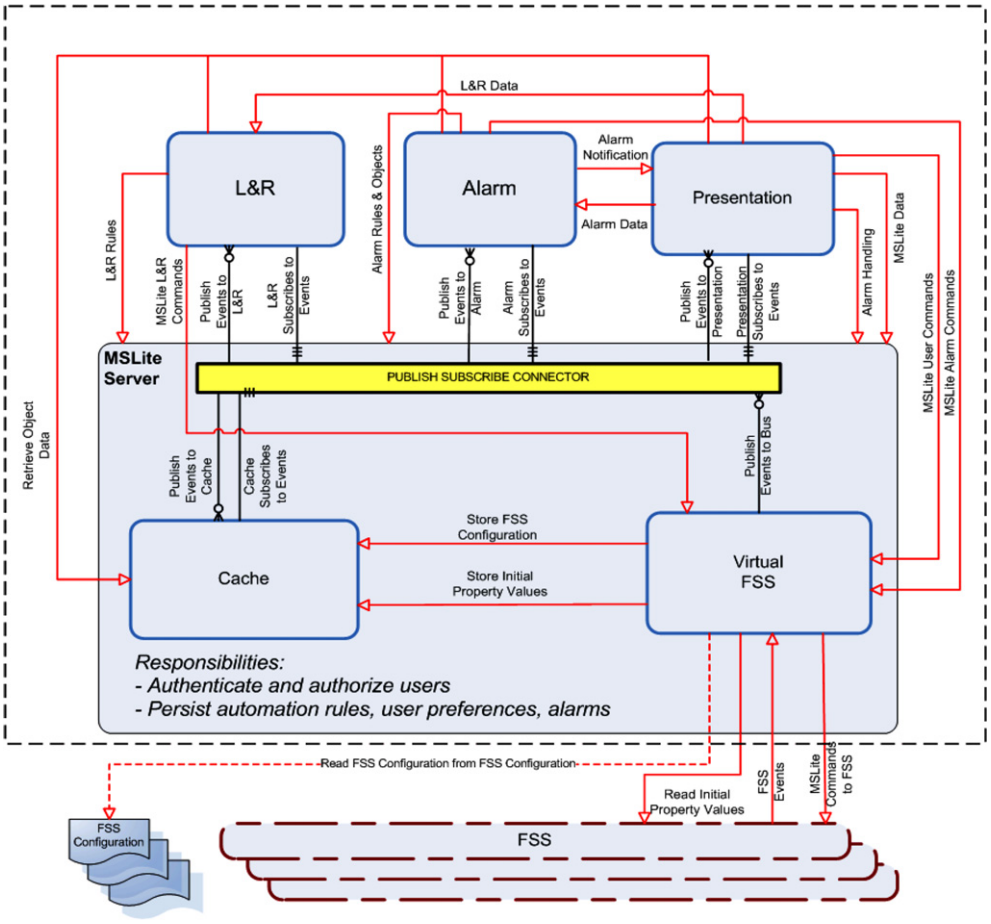

Fig. 15. Revisiting modifiability.

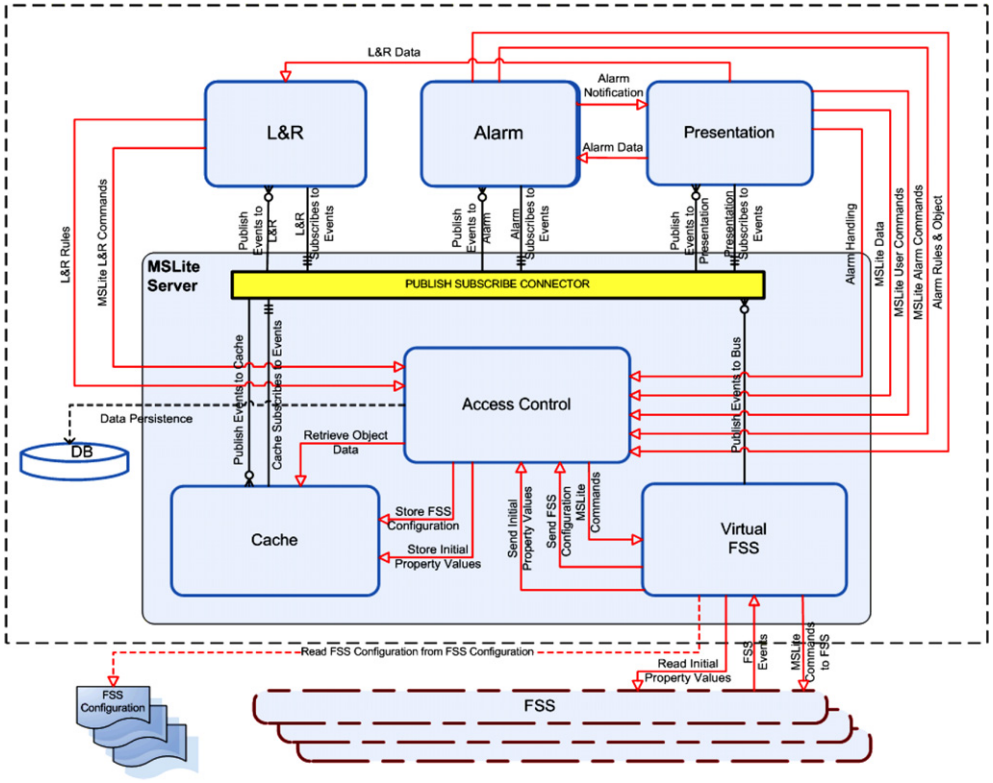

Fig. 16. Final architecture for MSLite system.

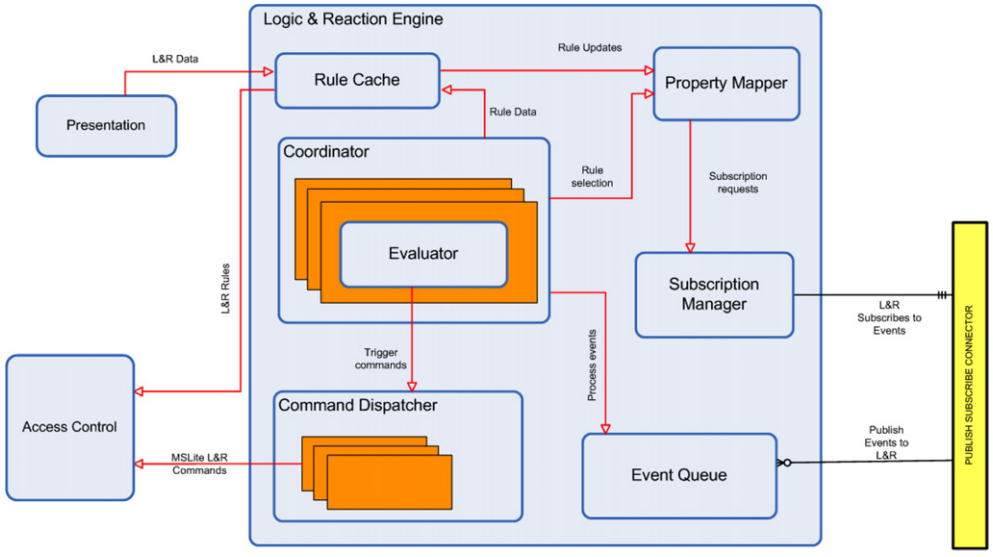

Fig. 17. Architecture for the logic and reaction engine of the MSLite system.

this level, it is possible to introduce more design tactics and patterns in iterative refinements. This is also the level at which we were able to use results from the OOAD analysis of the domain. To illustrate this connection, Fig. 19 shows the elements from the static structure of Fig. 18 which have associations with business domain types. These business types were derived from the business concepts model as illustrated in Fig. 4 and have a grayed background for differentiation.

The reference OOAD methodology we used so far offered a systematic process for identifying system and business interfaces and discovering their respective operations. At the specification level, a significant portion of these interfaces is independent of the architectural decisions since it derives mainly from the domain and requirement analysis. In our integrated approach, we relied on these interfaces for specifying the responsibilities of components and connectors.

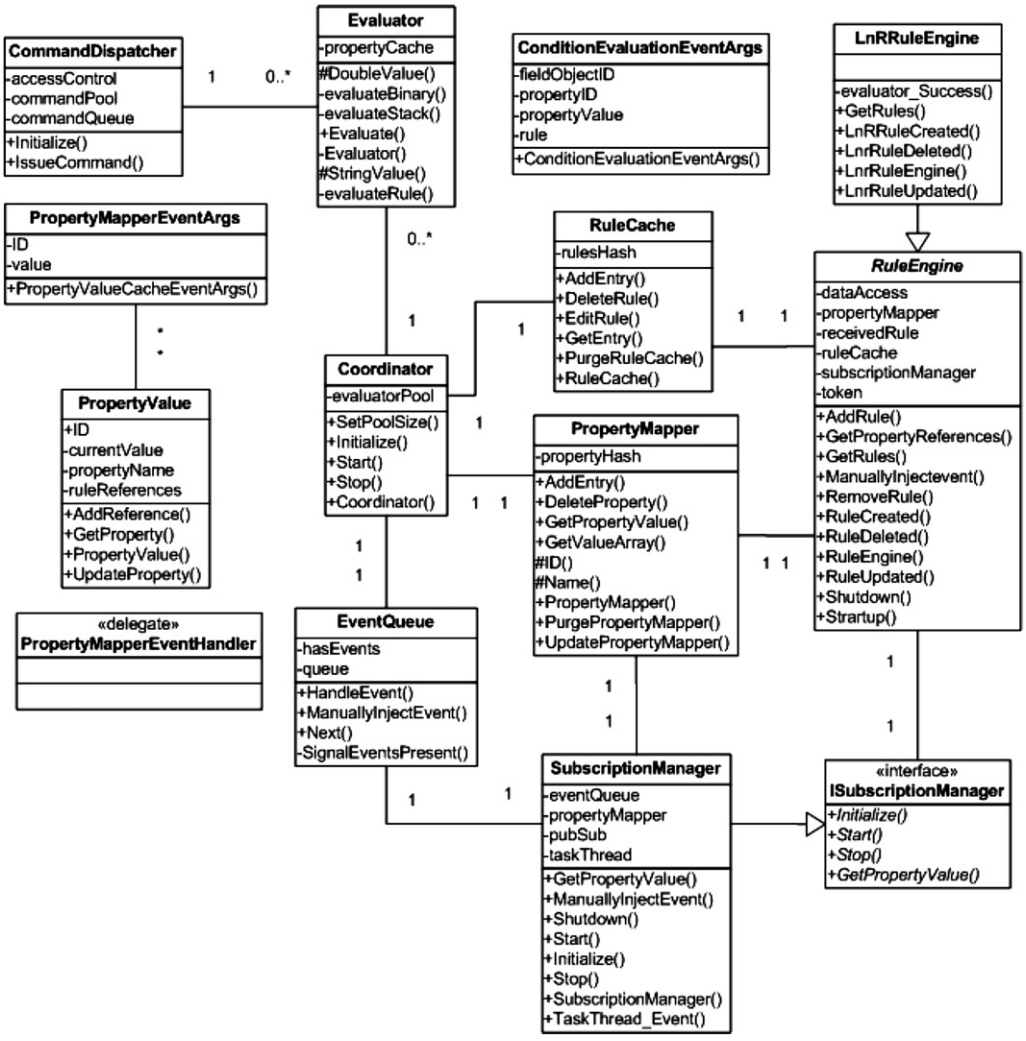

Fig. 18. Design class diagram for the logic and reaction engine.

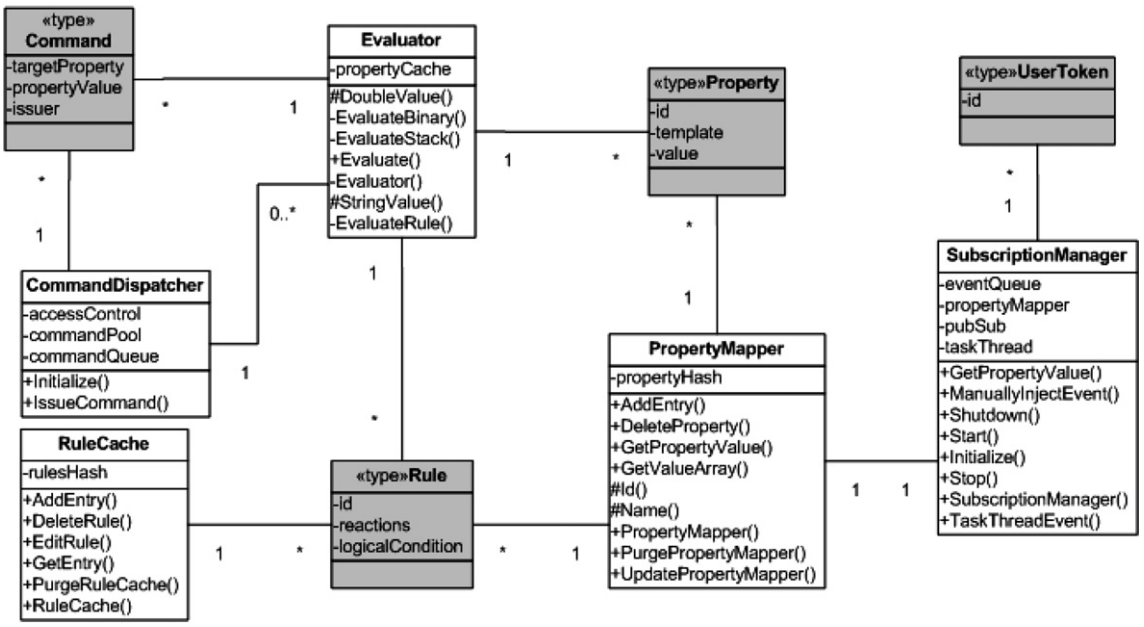

Fig. 19. Partial design class diagram for the logic and reaction engine, with business domain type associations.

The final output of the integrated approach was a component architecture with its related design decisions traceable to the business goals, a substantial specification for business and system interfaces, and a mapping of these interfaces to the components and connectors of the architecture.

Based upon the experiences of the Global Studio Project in implementing this integrated approach the following lessons-learned were reported (Mullick et al., 2006):

1. An architecture developed while requirements are changing is unstable, and for large-scale distributed development this instability results in frequent re-planning. For example, new dependencies arose such that one team required the definition of a portion of the object model under development by another team, sometimes at a later date. This disrupted the work plan and required significant architectural rework.
2. When work packages are determined from an architecture derived using OOAD the project task dependencies that arise from systemic properties such as memory footprint and performance budgets are not adequately investigated. This leads to duplicated work effort, conflicting solutions from different teams, and integration problems requiring rework.

To overcome these problems more upfront effort was required. To allow a greater understanding of temporal dependencies between tasks, and to generally improve understanding of the various dependencies between components (beyond functional dependencies) more centralized architectural development was instituted that addressed more concerns than merely functional requirements. Indeed, the very formal functional specifications from the first year were replaced with more textual requirements that augmented a more detailed architecture reflecting the new focus on systemic quality attributes.

6. Related work

The role of functional and non-functional requirements in achieving architectures fit for their intended purpose has been widely recognized. There is, however, a general lack of a design methodology that supports functional and non-functional requirements in an integrated manner (Cortellessa et al., 2005; Paech et al., 2002; Peraire et al., 1999; Robbins et al., 1998). A number of software architecture design methods, such as IBM's Rational Unified Process (Kruchten, 2004), Philips' BAPO/CAFCR (America et al., 2003), SEI's Attribute Driven Design (Bass et al., 2003) and Siemens' Four Views (Hofmeister et al., 2000), provide guidance for (macro) architecture analysis and design taking into account architecturally significant quality attribute requirements with minimal guidance for fine-grained (micro) architecture. We provide a summary of these methods in Table 4 (a detailed comparative analysis appears in Hofmeister et al. (2005)).

Of these, only the Rational Unified Process (RUP) has been used most widely in conjunction with OOAD (Larman, 2004). Unlike our approach, however, it relies on use case analysis for identifying architecturally significant requirements used in creating a baseline architecture for the system under design. While quality related information can be described along with use cases (Alexander, 2003; Zou and Pavlovski, 2006), to capture certain system level quality attributes that may precisely be the properties needed to design the architecture of the system may be difficult (Garlan, 2000; Kazman et al., 2004). For example, consider the following:

> "An improved COTS3 discrete event generator product is available for the system, and the system permits engineers to remove the old discrete event generator and incorporate the new one in less than two person-weeks."

While not impossible, expressing this requirement as a use case and elaborating it into system functions and their corresponding special requirements that help achieve this feature may be awkward.

RUP divides the development lifecycle of a software system into four phases – inception, elaboration, construction and transition. The baseline architecture is created during the elaboration phase at the beginning of which only a small fraction of use cases have been analyzed. Therefore, identifying architecturally significant use cases when most use cases are not fully understood can be challenging.

An additional criterion RUP uses for selecting use cases that are architecturally significant is based on whether or not a use case exercises most of the system under consider-

Table 4
Architecture analysis and design in software architecture design methods

Method	Architecture analysis	Architecture design
Rational unified process (Kruchten, 2004)	Identify use cases that are architecturally significant	Build an architectural prototype
BAPO/CAFCR America et al. (2003)	Identify elements in the business, process and organization context that are relevant to the architecture	Elaborate five CAFCR views, adding or refining artifacts (documents, models, code, etc.) suitable for a particular system
Attribute driven design (Bass et al., 2003)	Identify architectural drivers that stakeholders have prioritized according to business and mission goals	Recursively decompose the system using architectural patterns and tactics that satisfy the architectural drivers
Siemens four views Hofmeister et al. (2000)	Identify organizational, technological and product factors that influence the architecture	Make design decisions based on solution strategies identified for the influencing factors

ation. The problem with this is that early in the elaboration phase the system exists only as an evolutionary prototype.

Our work explores an approach dealing with functional and non-functional requirements in an integrated manner that draws on the strengths of the different design methodologies. We chose OOAD for its strengths in analysis and design of a system's micro-architecture and integrated it with ADD, an approach that most clearly articulates creation of the macro-architecture of a system by recursively applying architectural patterns and tactics. Since ADD requires architectural drivers as input but does not say how these are obtained, we chose QAW (Bachmann et al., 2002; Barbacci et al., 2000) for gathering architecturally significant requirements prioritized by stakeholders according to business and mission goals of the system under design.

7. Conclusions

Every system has a rationale for its creation. This rationale takes the form of business goals set forth by the organization creating the system and has a strong influence on the architecture of the system under consideration. In this paper we describe an architecture development process that takes these business drivers into consideration in the determination of the coarse-grained system components, and the appropriate separate of concerns, while still providing the necessary guidance to determine the fine-grained architectural detail. We do this by integrating architecture-centric methods that derive the systemic properties or quality attributes that guide the architecture of a system from stated business goals with a generalized OOAD approach.

The rationale for this combination is straightforward. OOAD strives for semantic closeness to the domain and seeks to maximize functional cohesion. To achieve these approaches focus on the principles of abstraction, encapsulation, information hiding and separation of concerns to define the structure of the system. In contrast, architecture-centric methods use architectural tactics associated with systemic properties as a guide for decomposing a system. When systemic properties are critical the architectures designed to maximize functional cohesion can fail.

To demonstrate the integrated approach, we described an example of its application in a multi-university global development study. The particular case analyzed was for a system called MSLite from the building automation domain. Through this case study we demonstrated the differences between OOAD and architecture-centric approaches, followed by the process and outcomes of the integrated approach that provides the benefits of both. Using architecture-centric methods we were clearly able to take into account the business goals and their related quality attributes in formulating a high level architecture. The analysis activities of OOAD provided a broad functional understanding of the system that served as input to the quality attribute scenarios used for the high level architecture. This architecture was then used as a basis for doing further detailed design and implementation using OOAD.

References

Alexander, I., 2003. Misuse cases help to elicit non-functional requirements. Computing & Control Engineering Journal 14 (1), 40–45.

America, P., Obbink, H., Rommes, E., 2003. Multiple-view variation modeling for scenario analysis. Proceedings of the Fifth International Workshop on Product Family engineering (PFE-5), Sienna, Italy. Springer-Verlag, pp. 44–65.

Bass, L., Clements, P., Kazman, R., 2003. Software Architecture in Practice, second ed. Addison-Wesley, Boston, MA.

Bachmann, F., Bass, L., Klein, M., 2002. Illuminating the Fundamental Contributors to Software Architecture Quality (CMU/SEI-2002-TR-025), Software Engineering Institute Carnegie Mellon University, Pittsburgh, PA.

Barbacci, M., Ellison, R., Weinstock, C., Wood, W., 2000. Quality Attribute Workshop Participants Handbook (CMU/SEI-2000-SR-001), Software Engineering Institute Carnegie Mellon University, July 2000, Pittsburgh, PA.

Cheesman, J., Daniels, J., 2001. UML Components: A Simple Process for Specifying Component-Based Software. Addison-Wesley, Boston, MA.

Cockburn, A., 2000. Writing Effective Use Cases. Addison-Wesley, Boston, MA.

Cortellessa, V., Di Marco, A., Inverardi, P., Mancinelli, F., Pelliccione, P., 2005. A framework for the integration of functional and non-functional analysis of software architectures. In: Proceedings of the International Workshop on Test and Analysis of Component Based Systems (TACoS 2004), 19, pp. 31–44.

Garlan, D., Software architecture and object-oriented systems. In: Proceedings of the IPSJ Symposium 2000, Tokyo, Japan, August 2000.

Hofmeister, C., Nord, R., Soni, D., 2000. Applied Software Architecture. Addison-Wesley, Boston, MA.

Hofmeister, C., Krutchen, P., Nord, R., Obbink, H., Ran, A., America, P., Generalizing a model of software architecture design from five industrial approaches. In: Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05), 2005, pp. 77–88.

Hohmann, L., 2003. The difference between Marketecture and Tarchitecture. IEEE Software 20 (4), 51–53.

Kruchten, P., 2004. The Rational Unified Process: An Introduction, third ed. Addison-Wesley, Boston, MA.

Jacobson, I., Booch, G., Rumbaugh, J., 1999. The Unified Software Development Process. Addison-Wesley, Boston, MA.

Kazman, R., Kruchten, P., Nord, R., Tomayko, J., Integrating Software-Architecture-Centric Methods into the Rational Unified: Process (CMU/SEI-2004-TR-011), Software Engineering Institute Carnegie Mellon University, 2004, Pittsburgh, PA.

Larman, C., 2004. Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development, third ed. Prentice Hall, Upper Saddle River, NJ.

Mullick, N., Bass, M., El Houda, Z., Paulish, D., Cataldo, M., Herbsleb, J.D., Bass, L., Sangwan, R., 2006. Siemens global studio project: experiences adopting an integrated GSD infrastructure. In: Proceedings of the IEEE International Conference on Global Software Engineering (ICGSE'06), Florianopolis, Brazil, October 16–19, 2006.

Neill, C.J., Laplante, P.A., 2003. Requirements engineering: the state of the practice. IEEE Software 20 (6), 40–45.

Paech, B., Dutoit, A., Kerkow, D., von Knethen, A., 2002. Functional requirements, non-functional requirements, and architecture should not be separated. In: Proceedings of the International Workshop on Requirements Engineering: Foundations for Software Quality, Essen, Germany, September 9–10, 2002, pp. 102–107.

Peraire, P., Riemenschneider, R., Stavridou, V., 1999. Integrating the unifed modeling language with an architecture description language. In: OOPSLA'99 Workshop on Rigorous Modeling and Analysis with the UML: Challenges and Limitations.

Robbins, J., Medvidovic, N., Redmiles, D., Rosenblum, D., 1998. Integrating architecture description languages with a standard

design method. In: Proceedings of the 20th International Conference on Software Engineering, Kyoto, Japan, April 19–25, 1998, pp. 209–218.

Schach, S.R., 2006. Object-Oriented and Classical Software Engineering, seventh ed. McGraw-Hill, Boston, MA.

Zou, J., Pavlovski, C.J., 2006. Modeling architectural non functional requirements: from use case to control case. In: Proceedings of the IEEE International Conference on e-Business Engineering (ICEB-E'06), pp. 315–322.