Toward Quality-Centered Design of Groupware Architectures

James Wu and T.C. Nicholas Graham

School of Computing, Queen's University, Kingston, Canada K7L 3N6
JamesWu@lincsat.com, graham@cs.queensu.ca

Abstract. Challenges in designing effective groupware include technical issues associated with concurrent and distributed work and social issues associated with supporting group activities. To address some of these problems, we have developed a *quality-centered* architectural design framework that links requirements analysis to architectural design decisions for groupware systems. The framework supports reasoned architectural design choices that are used to tailor software architecture to the unique quality and functional requirements of the software being developed. The framework has been applied to the development of the Software Design Board, a tool for collaborative software engineering.

1 Introduction

Groupware tools help people work and play together, providing integrated mechanisms for communication, collaboration and coordination [7]. Common examples of groupware include Lotus Notes' document respositories, the MSN Messenger instant messaging tool, the WebArrow/Conference online meeting tool, and the World of Warcraft massively multiplayer online game.

Groupware applications are difficult to construct, involving the difficult technological problems of supporting real-time interaction over a distributed system. A wide range of quality attributes affect the user's collaboration experience. Tools with poor *availability* may be unreliable and lead to inconvenience or loss of work. High *security* is required to ensure that the user's privacy is respected. Synchronous groupware requires high *performance* to support fluid interaction with other participants.

When translated into architectural choices, these requirements often conflict. For example, a requirement for high security might imply that all shared data should be stored at a single site, reducing the risk of unwanted data access. On the other hand, a requirement for high availability might imply that shared data should be replicated at multiple, redundant sites. Since there is no single groupware architecture that provides all of these qualities, architects of groupware systems must therefore carefully analyze their requirements to determine how to resolve these conflicts. *Architectural tradeoff analysis* involves the methodical comparison of architectural choices in order to determine what architecture best fits a system's requirements. Such analysis allows designers to reason about the properties of a system's implementation before it is developed, and as such is one of the fundamentals motivating architectural design.

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-3-540-92698-6_37

J. Gulliksen et al. (Eds.): EIS 2007, LNCS 4940, pp. 339–355, 2008.
© Springer-Verlag Berlin Heidelberg 2008

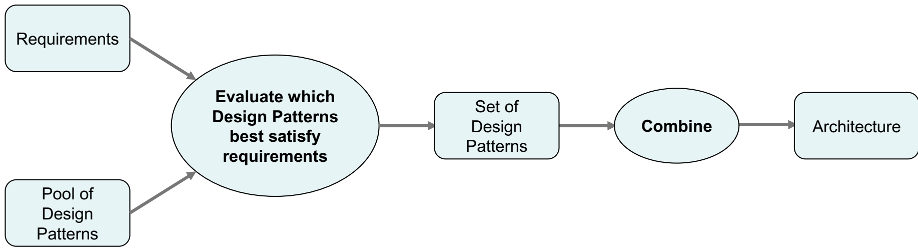

Fig. 1. How an architect applies quality-centered architectural design

To perform such analysis, designers require a set of alternative architectures from which their system may be composed, and a reasoning framework allowing them to assess the properties of each architectural choice. Such architectural "tool boxes" have not been widely developed.

In this paper, we present a quality-centered design framework for the groupware application domain. The framework consists of a set of architectural design patterns that can be combined to create groupware architectures, and a set of analytical models for quality attributes of interest to groupware. Architects can select those design patterns whose qualities best match the requirements of their groupware system, and combine them into an architecture.

The groupware domain provides a rich field of study for architectural tradeoff analysis, as there are numerous solutions to each architectural problem with no clear means of choosing between them. To illustrate its utility, we have applied our framework to the design and implementation of the Software Design Board [13], a tool supporting collaborative design of software systems.

2 Quality-Centered Architectural Design

We aim to improve users' experience with groupware applications through a novel quality-centered architectural design framework. The framework assists programmers in identifying candidate architectural styles for their groupware application, and in methodically determining which architecture best meets their requirements. Our contributions with the framework are:

- a set of *analytical models* that help relate software quality attributes to user experience,
- a set of *design patterns* that capture solutions to common problems in architecting groupware systems,
- a *quality impact matrix* that helps link the design patterns to desired system qualities.

Figure 1 shows how quality-centered architectural design links requirements analysis and architectural design, following the approach of Bass *et al.* [1]. Requirements are expressed in terms of key quality attributes such as performance, security, usability and availability. To help architects reason about design tradeoffs, our framework provides a *pool of architectural design patterns*, each of which embodies an architectural decision. In groupware, decisions might include

- whether to centralize or replicate shared data
- whether to use an optimistic or pessimistic concurrency control scheme
- how to reestabish service following the failure of a central communication hub
- how to distribute information required for awareness functions (such as telepointers).

The pool of design patterns includes different architectural solutions for these problems, representing different points in the space of tradeoffs. This provides architects with choices of how to best meet their application's requirements. The specification of a design pattern therefore includes analysis of its qualities, detailing the conditions where the pattern may improve (or worsen) the various quality attributes. For example, a pattern using an optimistic concurrency control scheme may improve feedback time while worsening the fidelity of different participants' views; a pattern involving data replication may improve the application's robustness to failure, while increasing its vulnerability to privacy violations.

The architect evaluates which design patterns best satisfy the application's requirements, and chooses a set of design patterns to be used in the architecture. These patterns must be combined to create an architecture for the system. This combination step may be straight-forward, but may involve further design work to enable the design patterns to work together. If combination of a set of patterns is not practical, new patterns may have to be chosen from the available pool.

In the following section, we examine a representative set of quality attributes, and develop analytical models which we will then use in section 4 to analyze our pool of groupware design patterns.

3 Qualities and Analytical Models

As seen in figure 1, architects select design patterns from a candidate pool based on their architectural qualities. Analytical models support this selection process, allowing the architect to evaluate design patterns with respect to a particular quality attribute. For example, *availability* is used to measure the frequency at which the system fails (and is unavailable for use); *security* measures how easily private data can be accessed by malicious third parties; *usability* measures how easily users can apply the system to performing their tasks; *functionality* measures how well the system matches the users' tasks; and *performance* measures how quickly the system responds to users' actions.

Analytical models serve as the basis for analyzing the qualities of design patterns. They provide a vocabulary for discussing quality attributes; for example, "performance" is computed from elements such as "local processing time", "network time" and "remote processing time", while "usability" of a groupware application comprises elements such as "fidelity", "consistency" and "awareness". Ultimately, analytical models allow us to determine the properties of architectural design patterns, supporting the choice of which design patterns best meet the requirements of a given application.

As representative examples, we now present analytical models for the availability, usability and performance quality attributes. These analytical models are developed specifically for the groupware domain. In section 4, we will show how these models allow us to precisely discuss the properties of design patterns.

In describing analytical models, we follow (but simplify) the approach of Bass *et al.* [1]. We specify an analytical model for each of a set of quality attributes as applied to the domain of collaborative applications. Analytical models are defined in terms of a set of *measures*, observable phenomena that influence the attribute of interest. For each analytical model, we then discuss what stimuli influence the measures, and give examples.

3.1 Analytical Model: Availability

Availability measures robustness of a groupware system in terms of what percentage of the time that the system is available for use. Poor availability leads to a negative user experience, as failures may lead to lost work or frustrating interruptions in collaborative sessions.

Analytical Model: Availability Domain: Collaborative applications Measures: Mean Time to Failure, Mean Time to Repair Details:

$$\text{availability} = \frac{\text{Mean Time to Failure}}{(\text{Mean Time to Failure} + \text{Mean Time to Repair})}$$

Where: *Mean Time to Failure* is the average length of time between component failures, and *Mean Time to Repair* is the average length of time required to restore the functionality of a failed component.

Discussion: In this context, Mean Time to Failure is influenced by both network and software component reliability. Any architectural feature than can improve the reliability of these components will increase the Mean Time to Failure experienced by individual collaborators. Architectural features that allow a component to remain functional in the presence of faults will increase the Mean Time to Failure. Similarly, features that influence the ability to reconfigure or repair the system when failures have occurred will affect Mean Time to Repair.

Examples:

1 Localizing the effects of any component failure can reduce Mean Time to Failure. For example, if a failure in a document sharing system can be localized, reducing the number of users who are unable to interact with the document, then the overall availability of the document to the group is increased.
2 Mean Time to Repair can be reduced by using redundant copies of core components to re-establish functionality in the event of a failure. This eliminates the processing associated with recovering the failed component, allowing functionality to simply be resumed by the back-up component.

3.2 Analytical Model: Usability

Using synchronous groupware should come as close as possible to the experience of collaborating in the same location. Usability measures aspects of how closely the groupware system achieves this goal.

Analytical Model: Usability

Domain: Collaborative applications

Measures: Fidelity, Consistency, Awareness

Details: *Fidelit*y measures the degree to which a participants view of shared artifacts represents their actual state. *Consistenc*y measures the degree to which different collaboration channels are synchronized. *Awarenes*s measures to what degree a participant can perceive the actions and attention of other participants.

Discussion: A primary source of reduced *Fidelit*y is the time that it takes for one participant's actions to be transmitted to other participants over a network. When participants are working asynchronously, their views of the system may become considerably out of date. Some algorithms for presenting participants consistent views of a shared state involve rollbacks of committed actions; in this case, Fidelity is compromised because the participant has been shown a view that is incorrect.

Groupware applications often allow people to collaborate using a variety of channels, such as voice, video, view of a shared artifact, and telepointers. *Consistenc*y measures how well these channels are synchronized. Poor consistency can lead to confusion, for example, a presenter talking over a slide that has not yet appeared on an audience member's display.

Groupware participants need to understand the activities and intentions of their collaborators. Such *awarenes*s may be improved via simple mechanisms such as telepointers, or advanced mechanisms such as gaze awareness.

Examples:

1. The use of an optimistic concurrency control algorithm allows a participant's actions to be reflected immediately in their view of a system. However, if this action conflicts with that of another participant, it may be rolled back. If conflicts are rare, the use of this optimistic concurrency control improves Fidelity by reducing feedback time; if conflicts are frequent, Fidelity is compromised due to high numbers of roll-backs.
2. Timestamping and buffering can be used to synchronize the data from different collaboration channels. This approach can improve Consistency, but at the cost of reducing Fidelity through increased latency.

3.3 Analytical Model: Performance

Performance affects the fluidity and naturalness of collaboration. If users find the tool to be unresponsive to their own actions or slow to report the actions of others, their experience of working together in a group will be negatively impacted.

Analytical Model: Performance Domain: Collaborative Applications Measures: Feedback Time, Feedthrough Time Details:

$$\text{Feedback Time} = \text{Local Processing Time}_{FB} + \text{Network Time}_{FB} + \text{Remote Processing Time}_{FB}$$

$$\text{Feedthrough Time} = \text{Local Processing Time}_{FT} + \text{Network Time}_{FT} + \text{Remote Processing Time}_{FT}$$

Where: *Loca*l *Processin*g *Tim*e is the time taken to process events at the initiating users local machine; *Networ*k *Tim*e is the time taken to transmit events across the network to remote machines, and *Remot*e *Processin*g *Tim*e is the time taken to process events received from the network at a remote machine.

Discussion: *Feedback Time* represents the time from a user performing an action to seeing the result of that action. *Feedthrough Time* represents the from a user performing an action to *other users*' seeing the result of that action. These measures are influenced by two factors – the performance of the network connecting collaborators (i.e., with respect to bandwidth and/or latency) and the amount of time required to process events before results can be displayed to users.

Examples:

1. Feedback Time can be reduced by eliminating the need for an event to be sent over the network before updating the display of its initiating user. That is, if the user's display can be updated without any network interchange, both the Network and Remote Processing times are removed from the above equation; i.e., (Feedback Time = Local Processing $Time_{FB}$.)
2. For Feedthrough Time, network traffic cannot be avoided; reducing the bandwidth required by events being sent across the network maximizes available bandwidth, thereby reducing the Network portion of the equation and therefore the overall time required.

The *performance* analytical model demonstrates how analytical models are developed for a particular application domain. The primary performance issues for groupware have to do with how quickly users see the results of their own actions (Feedback Time) and how quickly they see the results of others' actions (Feedthrough Time.) There are many other ways that performance of distributed systems can be measured (e.g., turnaround time, throughput, CPU load), but for groupware, Feedback and Feedthrough Time are the most important. By being able to concentrate on the measures that are most important for a particular domain, we can greatly reduce the complexity of analytical models.

4 Design Patterns

Following the approach of figure 1, developers of groupware applications first identify quality requirements, expressed in terms of the quality attributes discussed in section 3. The developer then selects from a pool of design patterns that best meet these requirements. The selected patterns are subsequently combined into a concrete architecture.

In order to architect groupware applications, we have identified a set of 21 design patterns supporting a range of groupware applications, involving real-time and asynchronous collaboration between co-located and remote collaborators. In addition to a description of how it is used, each design pattern is accompanied by an analysis summary. This summary explains the pattern's properties with respect to quality attributes, and is expressed relative to the relevant analytical models.

The design patterns shown in this paper are not intended to be comprehensive, but comprise a representative sample of the strategies that could be used to support synchronous groupware. As summarized in figure 2, the design patterns include support for:

- Both co-located and distributed interaction styles, including transitions between them;
- Both asynchronous and real-time interaction styles, including transitions between them;
- The creation of both syntactically correct and free-form artifacts, and the ability to seamlessly move between interactions with one style of artifact to the other;

	Performance	Availability	Usability
Document Replication: User's clients interact with a local copy of a shared document. Copies are synchronized to maintain an application-specific form of semantic consistency.	✓		
Centralized Document Processing: Users interact with a single remote copy of the shared document. Individual clients maintain local views of this remote data.	✓		
Star Topology: All client updates to a shared document are sent to a central hub for broadcast to the rest of the group.	✓	✓	
Mesh Topology: Every client broadcasts local updates directly to every other client.	✓	✓	
Localized Conflict Detection: Update events are broadcast to each client; the client is responsible for resolving conflicts as they occur, e.g., via operation transform.	✓		✓
Central Serialization with Migration: Update events are serialized before being broadcast to all clients, ensuring consistency concurrent updates.	✓		✓
Update Timeout: A ping/echo tactic allowing a client to determine whether an update has been received and processed by a server.		✓	
Dynamic Hub Migration: When multiple clients communicate via a hub located on one of the client nodes, the hub may migrate in case of failure of the hosting node.		✓	
Voting for Reconfiguration: A set of clients votes to decide whether to initiate reconfiguration of the topology.		✓	
Wait, Retry, Resync: A client that has detected a timed-out update briefly operates in "shadow" mode before attempting to update shared state again.		✓	
Event Broadcasting: Events affecting internal documents are forwarded to a central hub for broadcast to all interested clients.			✓
Event Broadcasting with Centralized Coordination: Events affecting external documents are forwarded to a central serializer before being broadcast to all interested clients.			✓
Distributed Directory Services: Each client maintains a directory of every other available client.			✓
Interface Awareness Cues: Each client implements interface features, such as telepointers, that support group awareness.			✓
On-Line Recognition: Free-hand sketches are sent to the recognizer as they are input, rather than batch-processed.			✓
Batch Recognition: Free-hand sketches are sent to the recognizer in batches.			✓
Plug-in Recognizers: Syntax recognizers have generic interfaces.			✓

Fig. 2. Extract from quality impact matrix: Summary of design patterns supporting the development of groupware architectures. Checkmarks indicate influence on quality attributes, either positive or negative.

- Both free-form and moderated interaction styles, including transitions between them;
- Interaction through a variety of devices, and movement between them.

To help designers navigate large numbers of design patterns, a *quality impact matrix* is provided (figure 2). This matrix shows the primary quality attributes that each pattern influences (either positively or negatively). Architects interested in improving a particular quality attribute can use the matrix to locate candidate design patterns for use in their architecture.

We now briefly describe two of the 21 design patterns that comprise the candidate pool compiled for the design of groupware tools. For both design patterns, we provide a brief description and an architectural diagram. The diagrams are based on the Workspace Architectural Model [12] (figure 3). The two selected design patterns show architectural alternatives that have equivalent functionality but markedly different influences on quality attributes. An architect would opt for one or the other based on the non-functional requirements specified for his/her particular project. Both design patterns address concurrency control.

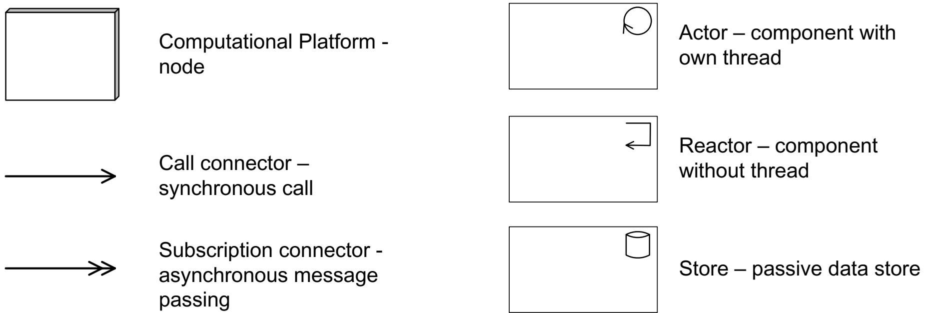

Fig. 3. The Workspace Architecture Notation used in figures 4, 5 and 7

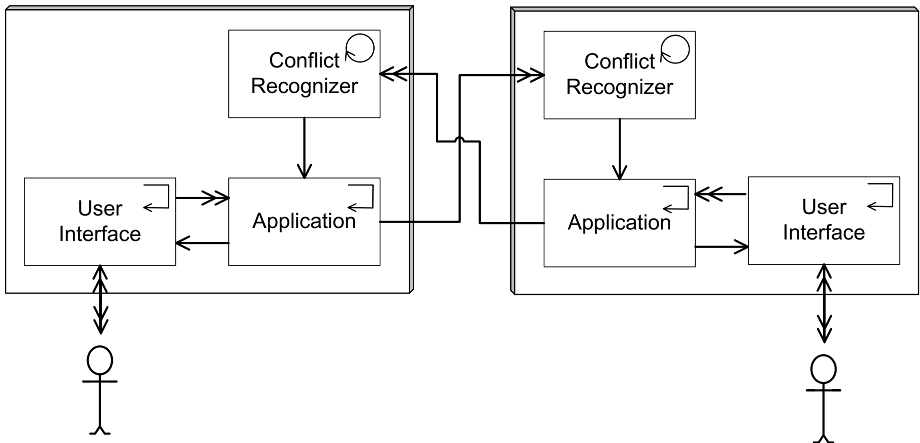

Fig. 4. Localized conflict detection design pattern

4.1 Localized Conflict Detection

The goal of *localize*d *conflic*t *detectio*n (figure 4) is to provide all participants in a groupware session with consistent views of shared state.

In this pattern, update events are broadcast to each client. Clients are responsible for detecting and appropriately resolving conflicts between different users updates. An appropriate implementation for this pattern could be operational transform [6].

Analysis Summary: This pattern influences both *availabilit*y and *usability*. Under availability (section 3.1), Mean Time to Repair benefits from the localization of the conflict recognizer. If one participant's node fails, the other participants can continue without problem, as they do not rely on the failed node's state or the state of its conflict recognition. Therefore, partial repair is quick.

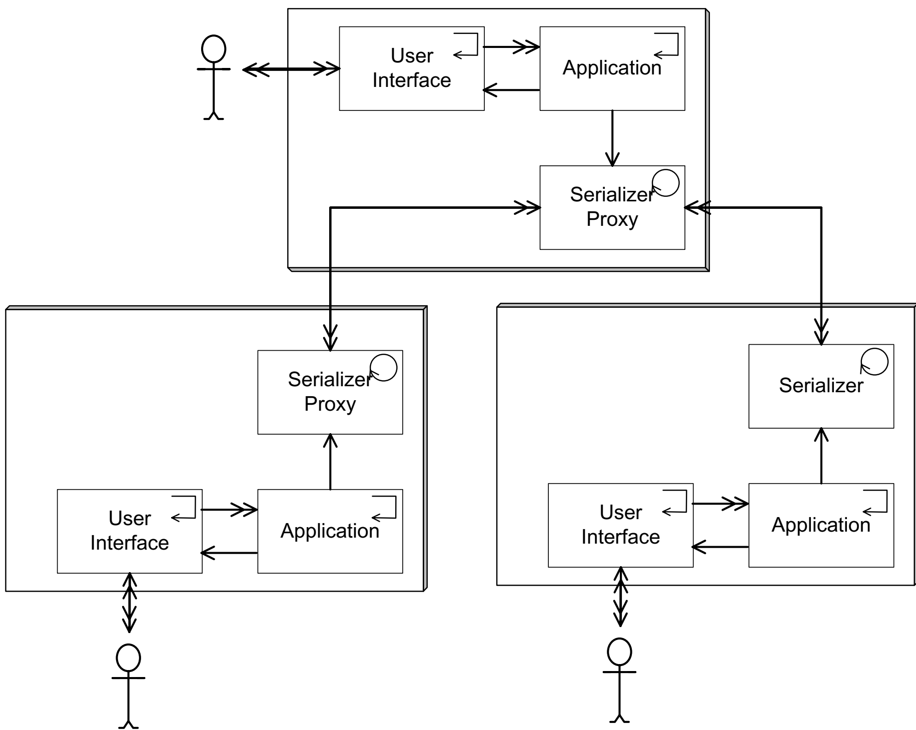

Fig. 5. Centralized Serialization with Migrating Serializer design pattern

Under usability (section 3.2), Fidelity may be improved or worsened by the adoption of this design pattern. Since the conflict recognizer is local, the results of participants' own actions may be shown immediately, without any need to send messages over the network. In the case of conflicts, however, the view may have to be rolled back. If conflicts between participants' actions are rare, Fidelity will be good; if conflicts are frequent, rollbacks will be frequent, having a negative effect on Fidelity.

4.2 Centralized Serialization with Migrating Serializer

As with the last pattern, the goal of *centralized serialization with migrating serializer* is to provide all participants in a groupware session with consistent views of shared state.

Update events are serialized before being broadcast to all clients. Since events are processed by each client in the same order, all users share a consistent view of the application's shared state. The component responsible for this serialization may migrate between client locations in response to patterns of update traffic. The architecture of this pattern is shown in figure 5.

Analysis Summary: This design pattern has an *availability* risk (section 3.1), particularly compared to Localized Conflict Detection. If the node hosting the serializer fails, then the system will be left in a bad state. A recovery algorithm would be required in order to choose a new node for the serializer.

Under *performance* (section 3.3), this pattern can increase Feedback Time relative to Local Conflict Detection because of increased Network times. However, this effect can be mitigated by migrating the serializer, reducing the average network delays experienced by all clients. Similarly, Feedthrough Time may be increased by this pattern due to contention at the centralized serialization component, or because migration of that component has increased the average Transmission Time between all clients. This pattern is particularly applicable to applications where only one user time performs input actions at a time, as the serializer will migrate to that user's computer.

Under *usability* (section 3.2), this approach has both negative and positive effects on Fidelity. Users on nodes with proxy serializers do not see the effects of their own actions until the action has been routed through a serializer on a different node, negatively impacting Fidelity. Conversely, the approach leads to no conflicts or rollbacks, positively affecting Fidelity.

4.3 Tradeoffs

The examples of the localized conflict detection and the centralized serialization with migrating serializer design patterns help illustrate the tradeoffs that developers must make when designing the architectures of groupware systems.

Localized conflict detection has excellent availability, and so is the better choice if good handling of partial failure is desired. Localized conflict detection provides good fidelity if conflicts are rare, but may be a poor choice if conflicts are frequent, leading to frequent undoing of users' actions. Centralized serialization is a good choice if conflicts are more frequent. However, centralized serialization may give poor feedback time; if $NetworkTime_{FB}$ is high (over a wide area network), this may be a poor choice. If most interaction is in the form of turntaking, then the migrating serializer will mitigate this problem.

In summary, therefore, localized conflict detection is a good choice when availability is important and conflicts are rare. Centralized serialization is superior if availability is less of a concern and if feedback time is unimportant (or clients are connected by a low-latency network.)

These examples illustrate the detailed analysis of architectural tradeoffs that is possible when design patterns are based on analytical models such as those of section 3.

5 Application: The Software Design Board

To gain experience with our quality-centered architectural design framework, we applied it to the development of *Software Design Board*, a tool supporting collaborative software design [13]. In section 5.1, we will show how our qualitycentered design framework was used to develop the Software Design Board and discuss its success.

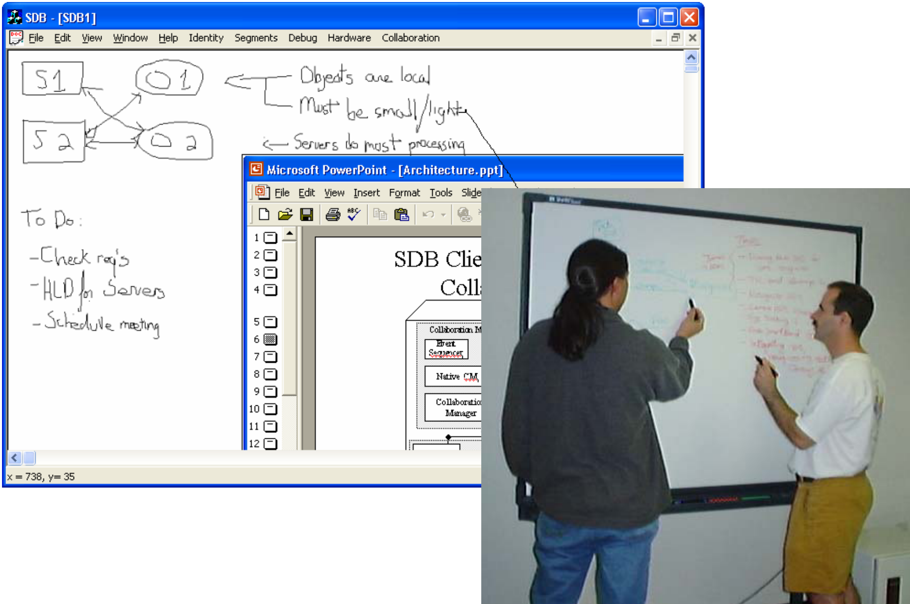

Fig. 6. The Software Design Board [13] permits free-hand drawing, automatic recognition of those drawings as structured diagrams, and supports collaborative use via electronic whiteboard or PC clients

The Software Design Board is a whiteboard-based, prototype tool intended to support collaboration in the early stages of software design. The tool supports a variety of styles of work helping in software design, and facilitates transitions between them. This is achieved be integrating informal media and flexible collaboration mechanisms, as well as supporting the migration between different software tools, devices and collaborative contexts. These facilities are intended to support fluid transitions between the some of the different styles of work in which we have observed software designers to engage [14].

As can be seen in figure 6, the core of the Software Design Board is its support for free-hand drawing and sketching, appropriate for brainstorming activities. Any number of people can participate in a brainstorming session from different locations, using either an electronic whiteboard or a traditional PC. Each participant sees the drawings of other participants in real-time. Telepointers allow participants to see where other participants are pointing. Gesture-based zooming and panning allows easy management of large drawing areas. Documents created with traditional programs such as Word or PowerPoint can also be embedded in the drawing area.

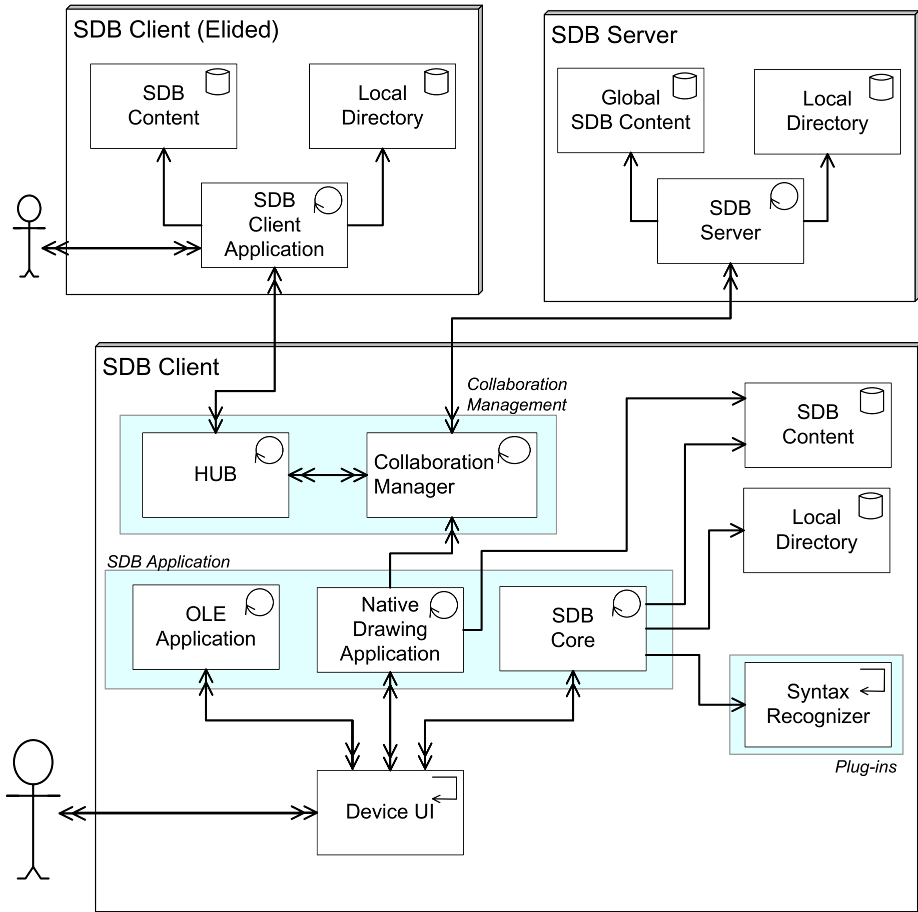

Fig. 7. Architecture of the Software Design Board

Free-hand drawings can be automatically converted to structure-drawings via a diagram recognition function, helping with the transition from rough sketches to formal documentation.

A participant can disconnect from the collaborative session (e.g., while traveling with a laptop), continue work, and merge his/her changes back when next reconnecting. If all participants disconnect, the state of the session is saved, allowing the next person to pick up where the session left off, using any device from any location.

The Software Design Board motivates quality requirements typical of groupware applications. It is important for partial failure to be handled effectively; if a participant's computer or network connection fails, the other participants should be able to continue uninterrupted. Security may be a significant issue, as design discussions may include sensitive data that should not be intercepted by malicious parties. Performance is important, as significant latency may inhibit the natural flow of discussion. And perhaps most importantly, the tool must enable natural collaboration, ensuring that participants easily understand the actions of other participants.

5.1 Architecture of the Software Design Board

In this section, we show which of the design patterns outlined in section 4 were selected and combined into the architecture of the Software Design Board.

The high-level architecture of the Software Design Board is described in figure 7. Each client application (*SDB Client Application*) maintains a local copy of all data (*SDB-Content*), as well as a directory of contact information (*Local Directory*) of people with ongoing collaborations. Each client application also interacts with a central server (*SDB-Server*), which maintains a global copy of all data. Additionally, the server maintains a global directory containing contact information for all clients in the system.

The *SDB Client Application* is expanded into four subsystems – *Collaboration Management*, *SDB Application*, *Plug-ins* and *Device UI*. The Collaboration Management Subsystem is responsible for managing shared data. The SDB Application Subsystem is responsible for the applications themselves, i.e., the native drawing application, control of external OLE applications and general functionality of the SDB itself (e.g. gesture interpretation.) The Plug-ins Subsystem maintains plug-in components, such as the free-hand drawing syntax recognizer. Finally, the Device UI Subsystem encapsulates the device-dependent user interfaces.

This architecture represents the composition of several of the design patterns summarized in figure 2:

- *Document Replication*: Each node maintains a local copy of its data (*SDB-Content*). The client applications (*SDB Client Application*) broadcast update events to each other in order to synchronize the distributed copies. This pattern was chosen over Centralized Document Processing for performance reasons.
- *Star Topology*: This is used to broadcast changes in the free-hand drawings to all session participants. The *Native Collaboration Manager* sends/receives events to/from a *Hub* component, which broadcasts those events to interested application components (other Native Collaboration Managers) in other nodes. Although this pattern has worse availability than the Mesh Topology, it was chosen to reduce the required number of network connections. The availability issue was addressed by the use of Dynamic Hub Migration, as described below.
- *Dynamic Hub Migration*: Within the star topology, a *Hub* is present on every node, facilitating migration of the broadcasting functionality between nodes. This pattern is effective when combined with the Star Topology pattern.
- *Distributed Directories*: Each node maintains a local directory of relevant peers (*Local Directory*). This directory is initially obtained from the server (*Global Directory*). Subsequently, clients directly broadcast relevant directory updates to each other in order to maintain current distributed directories without constantly checking the server for updates. A distributed directory has superior performance and availability to a centralized directory service.
- *Online Recognition*: The SDB performs structural recognition of hand-drawn diagrams. The application component (*SDB Core*) invokes the structure recognizer (*Syntax Recognizer*) before updating the local data

(*SD*B *Content*). This is performed for every update event received from the user interface.
- Online recognition was superior to Batch Recognition since it supports realtime feedback to the user.
- *Interfac*e *Awarenes*s *Cues*: A variety of interface awareness cues are implemented as part of the *SD*B *Core*, including telepointers and zooming/ scrolling functionality.

The central question in evaluating our experience with quality-centered architectural design is whether the requirements of the Software Design Board were met. The approach helped us to methodically assess which of a set of design patterns best addressed the application's requirements. The quality impact matrix helped in identifying the design patterns of interest. The analysis frameworks effectively provided a vocabulary for discussing the tradeoffs between patterns, allowing the choices summarized above. Once the application was built, its performance, usability and availability requirements were met as far as possible within a prototype tool.

The framework is a work in progress, and should be extended both to provide additional design patterns and additional quality attributes. Two new quality attributes of particular interest are security and development time. Security heavily influences how well an application respects the user's privacy, a question of enormous importance to groupware users. Estimates of development time place a significant reality check on architectural design, as the desired architecture may simply not be realizable within the available time or budget.

6 Analysis and Related Work

The work described in this paper builds extensively on earlier work in taxonomies of quality attributes [4, 8] and catalogues of the relationship between software architecture and quality attributes [1, 3]. These lines of research have attempted to identify architectural styles that achieve particular quality attributes. Additionally, there have been other systematic attempts to document the relationship between software architecture and quality attributes, including the Non-Functional Requirement Framework [5] and Attribute Driven Design [2].

Our experience with developing the Software Design Board leads us to a number of conclusions about Quality-Centered Design of software architectures.

First, we emphasize the importance of QCAD frameworks being domainspecific. If the domain is too broad, the framework developer will have an unreasonable number of design patterns to specify and analyze. Similarly, the complexity of the analytical models will grow, as a wide range of quality concerns need to be taken into account. It is practical to apply this approach if the domain is sufficiently narrow to keep the development of the framework tractible. Others have had success with domain-specific frameworks, most notably in the area of human-computer interaction [9] and IT systems [11].

The choice of design patterns to populate the framework is itself challenging. There is a constant tension between specifying many orthogonal design patterns with limited functionality versus fewer design patterns with more functionality. The former approach is more general, allowing design patterns to be more easily combined, possibly even in ways that the framework developer did not foresee. The latter approach

makes it easier for users of the framework to pick patterns of interest and combine them into architectures. Over all, making design patterns too fine-grained can lead to an explosion of patterns, while too coarse a granularity may make then hard to combine and may lead to important cases being missed.

Our experience shows that analytical models may be quantitative or qualitative. For example, our Availability and Performance models are based on measurable phenomena, while our Usability model is more subjective. Even with quantitative models, our reasoning is ultimately qualitative: it is difficult to provide a numeric value capturing the effect of a design pattern. There has been some progress in creating and validating analytical models in the groupware area [10] and in performance in general [11], but substantially more work is required. Of these approaches, we favour work that validates analytical models over approaches that require architects to do extensive mathematical analysis of their designs, simply in order to obtain results in a timely fashion. Particularly, as the required analysis becomes more complex, there is likely diminishing return on investment.

Nevertheless, the approach is useful now, as QCAD frameworks support methodical reasoning about the properties of software architectures. For groupware developers, even the experience of thinking about how quality attributes such as availability and security affect the user experience is highly beneficial. The framework as it stands already represents a significant advance over ad-hoc design.

Throughout our work, we gained experience in the development of QCAD frameworks, of which our groupware framework is one example. Figure 8 summarizes the steps required to create a new framework for a new domain. Our approach is similar to Bass *et al.*'s Attribute-Driven Design method, differing primarily in our use of design patterns as the unit of design, rather than ADD's more abstract tactics.

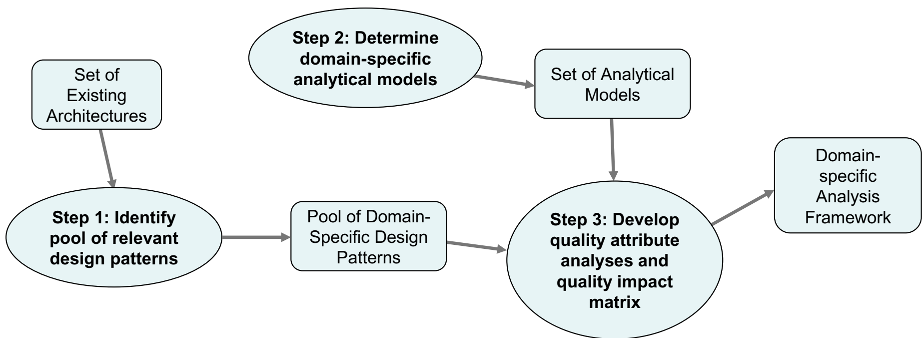

Fig. 8. How a framework developer populates a quality-centered design framework

A framework developer must first mine a set of existing applications to isolate useful design patterns, resulting in a *pool of domain-specific design patterns*. It is important to emphasize that each QCAD framework is specific to a relatively narrow domain, such as the development of groupware.

In order to help designers evaluate the tradeoffs between design patterns, *analytical frameworks* must be developed. The analytical frameworks are used to develop analytical advice associated with each design pattern, as well as a *quality impact matrix* used to help navigate the pool of patterns.

7 Conclusion

In this paper, we have presented a quality-centered design framework for groupware applications. This framework is an example of a more general approach in which domain-specific frameworks can be developed to help architectural design. We have illustrated the framework through its application to a significant groupware application, the Software Design Board. We have shown how the Software Design Board is constructed by combining design patterns suggested by our QCAD framework.

Acknowledgements

This work benefitted from the generous support of the Natural Science and Engineering Research Council of Canada, the Ontario Centres of Excellence, and the Network for Effective Collaboration Technologies through Advanced Research.

References

1. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice, 2nd edn. SEI Series in Software Engineering. Addison-Wesley, Reading (2003)
2. Bass, L., Klein, M., Bachmann, F.: Quality attribute design primitives and the attribute driven design method. In: Software Product-Family Engineering. LNCS, pp. 169–186. Springer, Heidelberg (2001)
3. Bergey, J., Barbacci, M., Wood, W.: Using quality attribute workshops to evaluate architectural design approaches in a major system acquisition: A case study. Technical Report CMU/SEI-2000-TN-010, Software Engineering Institute (2001)
4. Boehm, B., Brown, J., Kaspar, H., Lipow, M., McLeod, G., Merrit, M.: Characteristics of Software Quality. North Holland, Amsterdam (1978)
5. Chung, L., Nixon, B., Yu, E., Mylopoulos, J.: Non-Functional Requirements in Software Engineering. Kluwer Academic Publishers, Dordrecht (2000)
6. Ellis, C., Gibbs, S.: Concurrency control in groupware systems. In: Proceedings of the ACM Conference on the Management of Data (SIGMOD 1989), Seattle, WA, USA, May 2–4, pp. 399–407. ACM Press, New York (1989)
7. Ellis, C., Gibbs, S., Rein, G.: Groupware: Some issues and experiences. Communications of the ACM 34(1), 38–58 (1991)
8. International Organization for Standardization, International Electrotechnical Organization. International Standard ISO/IEC 9126. Information technology – Software product evaluation – Quality characteristics and guidelines for their use
9. John, B., Bass, L., Segura, M., Adams, R.: Bringing usability concerns to the design of software architecture. In: Proc. Engineering Human-Computer Interaction/ Design, Specification and Verification of Interactive Systems. LNCS, pp. 1–19. Springer, Heidelberg (2004)
10. Junuzovic, S., Chung, G., Dewan, P.: Formally analyzing two-user centralized and replicated architectures. In: Proc. ECSCW 2005, pp. 83–102. Springer, Heidelberg (2005)
11. Koziolek, H., Firus, V.: Empirical evaluation of model-based performance prediction methods in software development. In: Quality of Software Architectures. LNCS, pp. 188–205. Springer, Heidelberg (2005)

12. Phillips, W.G., Graham, T.C.N., Wolfe, C.: A calculus for the refinement and evolution of multi-user mobile applications. In: Gilroy, S.W., Harrison, M.D. (eds.) DSV-IS 2005. LNCS, vol. 3941, pp. 137–148. Springer, Heidelberg (2006)
13. Wu, J., Graham, T.C.N.: The Software Design Board: A tool supporting workstyle transitions in collaborative software design. In: Proc. Engineering Human-Computer Interaction/ Design, Specification and Verification of Interactive Systems. LNCS, pp. 363–382. Springer, Heidelberg (2004)
14. Wu, J., Graham, T.C.N., Smith, P.: A study of collaboration in software design. In: 2003 International Symposium on Empirical Software Engineering (ISESE 2003), Rome, Italy. IEEE Computer Society, Los Alamitos (2003)

Questions

Prasun Dewan:
Question: Does your work allow for optimization of combinations of parameters/ Foe example, high awareness compensates for low consistency management. This is an apparent trade-off, but not a real one, as the usability does not degrade because of low consistency.

Answer: The user will simply pick an architecture with high awareness and low consistency management.

Laurence Nigay:
Question: Would it be possible that design patterns re not compatible?

Answer: It is a loop mechanism, back to the quality factor.

Phil Gray:
Question: This approach is based on the identification of requirements which drives the analysis and assessment. However, requirements are subject to change. How would/could you handle this fact?

Answer: Requirements always subject to change. Basically, we should always do the best we can to anticipate potential change and design with that in mind.

Question: What about "malleability" or "support for change" as a quality attribute for an architecture?

Answer: Yes. We don't have that, but it would be a great idea.