FLEXIBLE SHARING OF DISTRIBUTED OBJECTS BASED ON PROGRAMMING PATTERNS

by
Vassil Roussev

A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill
2003

Approved by

_____
(Advisor: Prasun Dewan)

_____
(Reader: Hussein Abdel-Wahab)

_____
(Reader: Ketan Mayer-Patel)

_____
(Reader: F. Donelson Smith)

_____
(Reader: David Stotts)

ABSTRACT

Distributed collaborative applications allow a group of physically dispersed users to work on a common task. To simplify and lower the cost of developing such applications, a large number of collaborative sharing infrastructures have been built. A common approach employed by sharing infrastructures is to present the programmer with shared programming abstractions as a basis for implementing multi-user applications. A shared abstraction extends a traditional single-user abstraction, such as an object, with automatic collaboration support.

In this thesis, we summarize the achievements and limitations of current approaches to implementing shared abstractions and argue that they have not been successful in *simultaneously* addressing the issues of automation, abstraction flexibility, sharing flexibility, extensibility, and legacy code reuse. We present a new infrastructure model designed to fulfill these requirements better than existing systems. At the core of the model is the notion of a *programming pattern* based on which we define a new shared abstraction model. A programming pattern is a consistent naming convention that allows the logical structure of an object to be implicitly derived from its observable state. We define a pattern specification language that allows programmers to formally express the patterns they use and show that the set of supported pattern-based abstractions subsumes the abstraction models of existing systems.

We complement our abstraction model with an architectural model that allows the practical use of patterns in the application development. Furthermore, we introduce a sharing model that subsumes and provides a descriptive mechanism for specifying application layering.

We describe a prototype implementation of our conceptual model, as well as our experience in developing collaborative applications with it. We also discuss several problems in which we have successfully employed a pattern-based approach outside the domain of collaboration. Next, we present a requirement-by-requirement evaluation of our work relative to existing systems showing that, overall, our system performs better in satisfying the

requirements. Finally, we present our conclusions and outline the directions in which we plan to extend this work in the future.

Keywords: distributed collaborative applications, computer supported cooperative work, CSCW, programming patterns, distributed object sharing

ACKNOWLEDGEMENTS

First and foremost, I thank my advisor, Prasun Dewan, for his time and patience, his persistent encouragement, and his numerous lessons in research and writing, without which this dissertation would not have been possible.

I gratefully acknowledge the support of my committee members, Hussein Abdel-Wahab, Ketan Mayer-Patel, Don Smith, and David Stotts, whose comments have helped me in shaping the contents and improving the presentation of this work. I would like to thank the entire Department of Computer Science at UNC for their persistent support and for creating the wonderful academic environment that I have had the privilege to be a part of.

Last but not least, I would like to thank my parents, Radka Kanceva and Rossen Roussev, for their support and encouragement and for their lessons in life that have helped me in everything I have achieved.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

Distributed collaborative applications (a.k.a. multi-user applications, or groupware) allow a group of geographically dispersed users to work together on a common task, such as editing a document. Most often this is achieved by presenting users with the abstraction of a shared artifact, which allows multiple users to modify it and to observe the effects of each other's actions. An example of a shared artifact is given on Figure 1.1, which shows two users—*A* and *B*—working together on a shared outline of a paper using a multi-user outline editor. In this case, the application presents both users with the same view of the outline thereby making them instantly aware of each other's actions. Another possibility is that *A* and *B* might be disconnected for a period of time and would like to keep working on separate versions and be able to merge them automatically. As another example that falls somewhere in between, consider the case where *A* and *B* are updated periodically of each other's progress, or only when changes of a given (user-defined) level of significance are introduced.

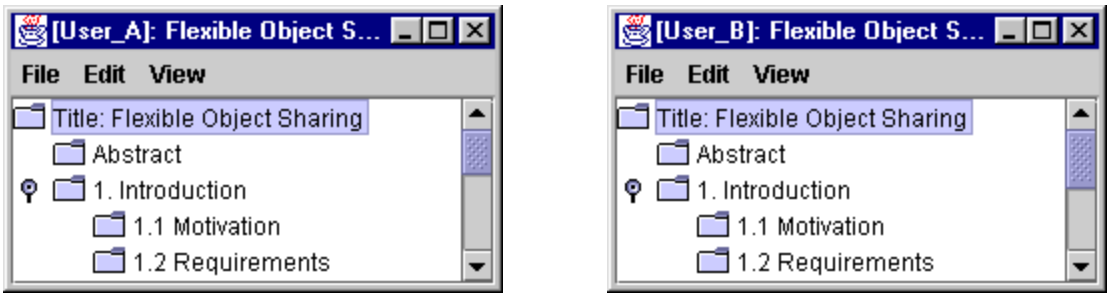


Figure 1.1 Users A and B working together on a shared outline

In general, there is a whole spectrum of useful sharing scenarios and, ideally, a collaborative application should be able to *dynamically* adapt to all the scenarios that are relevant to its domain. Experience shows that developing such applications from the ground up is a costly and error-prone process. Therefore, a number of software infrastructures, which we refer to as *sharing*, or *collaborative infrastructures*, have emerged to facilitate the development effort by providing reusable mechanisms for distributed object sharing that are applicable to large classes of applications.

However, the sharing mechanisms offered by existing infrastructures have not been designed to completely meet the needs of collaborative applications. The goal of this research is to analyze the limitations of existing approaches and to develop a sharing infrastructure that better addresses the requirements of multi-user applications. We have taken the following steps to systematically achieve this goal:

- Identify generic usage scenarios relevant to a wide range of collaborative applications;
- Based on the scenarios and general software engineering principles, derive a set of generic infrastructure requirements;
- Analyze the achievements and shortcomings of existing solutions with respect to the requirements;
- Based on new mechanisms, develop an infrastructure that better meets these requirements;
- Evaluate the result of our work by developing actual collaborative applications and comparing it with respect to the requirements and existing solutions.

The rest of the presentation successively discusses the above points starting with the description of three basic sharing scenarios.

1.1 Collaborative Sharing

Since it is nearly impossible to exhaustively enumerate all plausible sharing scenarios from a user's point of view, we classify them into three broad categories based on a simple conceptual model of collaborative applications. Later on, in Chapter 3, we will refine and formalize this model to enable its incorporation into our sharing infrastructure.

Generally, a regular single-user application works by taking user input, performing some computation, and displaying back the results to the user. It also allows the user to persistently store and retrieve the results of the computation. Therefore, as Figure 1.2a illustrates, we can view an application as consisting of three basic functional layers: *user interface* (UI), which directly interacts with the user, *core application*, which performs the application-specific computation, and a *repository*, which is responsible for persistent storage. Central to this model is the core application layer, which maintains an abstract representation of the virtual

artifact (or *object*), which is manipulated through the user interface, and is kept persistent by the repository.

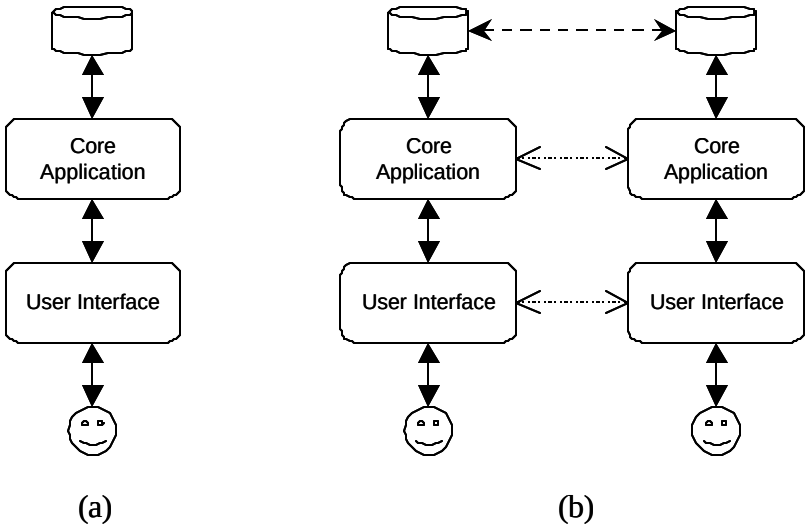


Figure 1.2 Basic Layer-based Application Decomposition

To extend this model to the domain of distributed collaboration, we add sharing functions to one, or more, of the application's layers as shown on Figure 1.2b. The vertical arrows, as in the single user case, represent the flow and transformation of events that are independent of the number of users and, thus, represent the baseline single-user semantics of the application. On the other hand, horizontal lines represent exchange of state information, or *sharing*, between peer application layers. The figure shows a *logical* view of the sharing process and does not necessarily imply that the layer is physically replicated, or that layers communicate directly with each other. For example, sharing in distributed file systems, such as *NFS*, is achieved by users accessing the same physical copy of the file. However, we can still apply the *logical* view of the sharing shown on Figure 1.3.

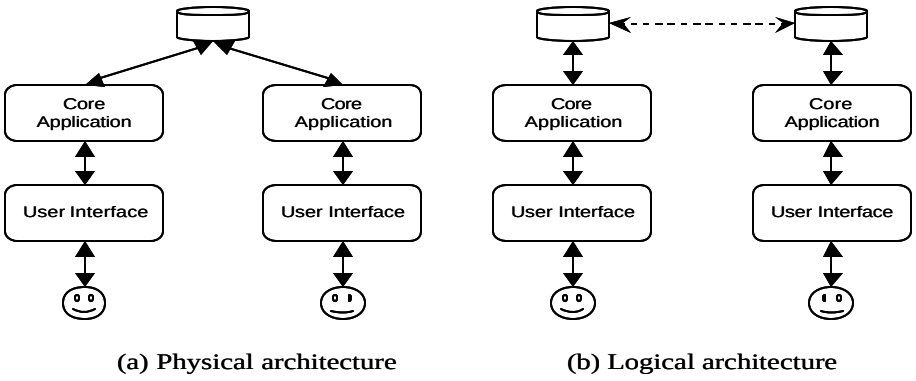


Figure 1.3 Physical vs. logical view of sharing

In any case, since users are physically distributed, at least (part of) the user interface layer must also be distributed. Given this simple model, let us now consider the different scenarios that can be supported through sharing at each of the three levels.

1.1.1 User interface-based sharing

User interface-based sharing allows collaborators to work together by interacting with the object through the same user interface (UI) presentation in multiple computers as shown on Figure 1.1. This is commonly referred to as *what-you-see-is-what-I-see* (WYSIWIS) *sharing*. For this scheme to be effective, users must be able to modify the object and observe each other's modifications in real time, or *synchronously*. WYSIWIS also implies that, at any given moment, there can be only one user who interacts with the application. Otherwise, users might issue conflicting commands concurrently and the WYSIWIS abstraction becomes undefined. Thus, WYSIWIS is useful in cases, such as a moderated editing session, or a remote presentation, where the restriction of one active user is either desirable or acceptable.

UI-based sharing is largely limited to WYSIWIS because the sharing mechanism has no knowledge of the shared object. Consequently, if user views of the object are allowed to diverge, this could lead to separate versions with no means to restore consistency. The only way to relax the WYSIWIS requirement is to allow some aspects of the UI that do not affect the state of the object (such as formatting parameters) to be different for different users. However, such information is inherently application-specific and implementing an

application-specific solution would defeat the main advantage of UI-based sharing—complete automation.

Complete automation here refers to the fact that a single infrastructure can provide sharing for large classes of single-user applications at no additional development cost. This is possible due to the fact that, currently, virtually all applications are developed using standard user interface components provided by the basic underlying infrastructure—the window system (e.g., X Windows) and/or software development toolkits (e.g., *Java AWT/Swing*). Thus, it is sufficient to replace the original single-user implementation of the standard UI components with a collaborative one and all applications using the basic infrastructure become UI-shareable.

1.1.2 Repository-based sharing

Another approach to complete automation is to use repository-based sharing. It is based on users taking turns at working on and updating a (logically) single persistent copy of the shared object. For example, users could use a distributed file system, which gives them remote access to the same physical file over the network. To exchange state, users must explicitly commit any updates made to the shared object to make them visible to other users, which in turn must explicitly request the current version in order to observe the update.

In many respects, this sharing scenario has the opposite characteristics of the previous one—users are not restricted in their concurrent access to the object, they may have completely different views of it, and their updates are propagated *asynchronously*, i.e. there is no limit on the interval between the time one user makes a change and the time it becomes visible to others. Asynchrony is an inherent trait of repository-based sharing—due to the large overhead of communicating with secondary storage, it is impractical to use it as a medium to communicate user updates at interactive rates.

Since users may work on the shared object independently, this may lead to the existence of several different (and potentially conflicting) versions of it. Therefore, the task of a collaborative application is to provide a suitable solution that either helps users avoid the problem altogether or helps them resolve inconsistencies in an automated fashion.

1.1.3 Application-based sharing

Application-based sharing allows the automated exchange of updates to the (in-memory) abstract representation of the shared object. For example, our outline document can be represented as a record consisting of a title (a string) and a sequence of sections. In turn, each section is also a record with a heading and a (potentially empty) list of subsections. Despite the fact that the structure of the shared object is relatively simple, the number of useful sharing scenarios is non-trivial even for a small outline document.

Let us first consider the issue of *what* should be shared. In real-world collaboration, it is quite common for collaborators to split the work among themselves and work individually, or in small groups on different parts of a document. Hence, to be relevant, a collaborative application should at least be able to replicate this model of collaboration. For example, it should be possible for users A and B to share Section 1 of the outline, which they have agreed to work on together, but not to share the rest as these will be worked on individually. More generally, each and every node of the outline's tree structure is potentially shareable so, ideally, it should be possible to control the sharing of the outline at any level of its hierarchy. For that purpose, the logical structure of the shared object must be known to the sharing mechanism. This is not a problem for custom-built collaborative applications, however, for sharing infrastructures this is a challenge because the set of supported abstractions is determined at design time. Therefore, an infrastructure should support the sharing of as many abstractions as possible and should be designed in a way that allows the set of abstractions to be easily expanded. We refer to the set of abstractions that an infrastructure can automatically share as *abstraction flexibility.*

Another issue to consider is the question of *when* to share an object. That is, under what condition(s) the exchange of updates from different users will take place. Two obvious choices are the synchronous and asynchronous sharing presented in the previous sections. However, there is a whole spectrum of alternatives in between that can be quite useful. For example, users may want synchronous collaboration but because of network overload, this may not be possible. In that case, users may consider *periodic* updates where exchange takes place once per user-specified time interval that can range from seconds to days and could be adjusted depending on network traffic. Even if communication delay is not an issue, some users may not want to be constantly distracted by being forced to observe every incremental

change of their team members but still want to be informed of "important" updates. While the notion of importance is somewhat subjective, we can consider proxy criteria for the exchange based on some measure of correctness, or completeness of the user actions. For example, send the new value of a text field only after the user has indicated he is done editing it (by moving to the next item), or after an explicit commit command.

Finally, each user may want to control the way he is sharing with each individual or group of participating users [Colab, Suite]. Thus, user *A*, for example, may want to share synchronously Chapter 1 with user *B*, while exchanging only hourly updates with *C* on Chapter 2. Furthermore, user preferences for sending and receiving updates may be asymmetric. For example, a user might be willing to send out every incremental change he is performing but might only want to receive updates from others that are explicitly committed.

In summary, there is a wide variety of application-sharing scenarios but they can be classified based on three major parameters: *what* is shared, *how* is it shared, and *with whom* is it shared. For the rest of this thesis, we refer to a particular combination of the values of these three parameters as a *sharing mode*. By exhaustively enumerating all sharing modes supported by a particular collaborative application or infrastructure, we can get a measure of its *sharing flexibility.*

Based on our discussion so far, as well as some basic software engineering principles, we can identify the following collaboration infrastructure requirements along with the respective criteria to measure the extent to which any given infrastructure satisfies them. For each requirement we also provide examples of systems (or classes of systems) that provide maximum and minimum support, respectively.

1.2 Infrastructure Requirements

1.2.1 Automation

In general, we can distinguish between two types of software infrastructures—enabling and automating. The main purpose of the former is to make possible the development of a new class of applications, whereas the latter seeks to automate the development process of an existing class of applications. In the case of distributed collaborative applications, we can consider TCP/IP sockets to be the basic enabling infrastructure upon which all such

applications are built. Therefore, we are interested in comparing infrastructures whose primary goal is the automation of collaborative applications.

Typically, the level of automation provided by an infrastructure is measured by comparing the amount and complexity of the code a programmer has to write to build a typical application with and without the infrastructure. In the case of a collaborative infrastructure, it is a measure the developer's effort required to achieve multi-user behavior with respect to the single-user case.

In this respect, TCP/IP provides the lowest level of automation because it only provides a basic point-to-point reliable ordered communication mechanism. The application must deal with issues of multicasting to a dynamic group of users, defining a protocol for encoding/decoding of updates to the shared object (e.g., insertion of a new section), and coordinating concurrent updates submitted by users. On the other end of the spectrum, shared window systems, such as XTV [1], provide the most automation because they require no development effort to convert regular applications into collaborative ones.

1.2.2 Flexibility

The flexibility offered by a sharing infrastructure is defined by its ability to support a variety of collaborative applications and scenarios. Conceptually, the requirements of automation and flexibility present a fundamental trade-off—the more an infrastructure assumes about the supported applications and scenarios, the more automation it can provide. Inversely, the less it knows about the application and the usage scenarios, the wider the range of applications and scenarios it can accommodate (at a higher relative cost). For example, shared window systems (discussed in detail in the next chapter) provide fully automatic sharing by assuming one particular sharing mode—WYSIWIS, whereas TCP/IP sockets place virtually no restrictions on the sharing while providing low automation.

Therefore, to enable meaningful comparisons among different infrastructures' flexibility features we must first choose the level of automation at which the infrastructures are compared. In this work, we consider infrastructures that provide *high* level of automation for at least one of the scenarios discussed in the previous section.

We distinguish among three aspects of flexibility to better characterize different systems.

- **Sharing flexibility.** An infrastructure should automatically support as many modes of sharing as possible. In other words, a small amount of development effort should be required to build an application that supports a variety of sharing modes. To compare two infrastructures—*A* and *B*—with respect to sharing flexibility, we compare the range of sharing modes they support. If *A* supports all the sharing modes of *B* and, in addition, supports modes that are not supported by *B*, then *A* has a higher (degree of) sharing flexibility.

  Furthermore, specification of the sharing modes should follow the logical structure of the shared object. In other words, it should be possible to specify different sharing policies for logically autonomous parts of the object. For example, it should be possible to provide separate sharing specifications for the sharing of the title of the outline and the sharing of each of the outline's sections.

  Among existing systems, shared window systems are a good example of an infrastructure that provides no sharing flexibility—only WYSIWIS is supported, whereas Suite provides the most sharing flexibility through a parameterized sharing mechanism which systematically covers a whole range of useful sharing modes.
- **Abstraction flexibility.** As part of the task to lower the development cost, an infrastructure must support automatic sharing of as many types of objects as possible. Most infrastructures offering high automation provide at least one shared abstraction. However, providing only a fixed set of shared abstractions inherently limits the applications that the infrastructure can support. Therefore, the infrastructure should support programmer-defined shared abstractions that reflect the specific needs of every application. Specifically, it should support:
  - **Programmer-defined semantic objects.** Semantic objects maintain the most abstract representation of the object manipulated by the end user and, therefore, the best understanding of his intentions. Therefore, sharing of semantic objects offers the most flexibility in accommodating concurrent updates from a group of users while preserving their intentions. Inherently, semantic objects are application-specific and, therefore, the infrastructure should be able to share as many programmer-defined semantic objects as possible.

- **Programmer-defined user interface.** For many applications, such as a drawing editor, the custom features of the graphical user interface through which the semantic object is manipulated are just as important as the semantic object itself. Therefore, the infrastructure should support the sharing of programmer-defined UI and the sharing of the semantics object should be independent of the implementation of the user interface.

To compare two infrastructures—*A* and *B*—with respect to abstraction flexibility, we compare the range of sharing abstractions they support. If *A* supports all the shared abstractions of *B* and, in addition, supports abstractions that are not supported by *B*, then *A* has a higher (degree of) abstraction flexibility.

As an example of systems with no abstraction flexibility, consider shared window systems—they provide only a single abstraction, a shared window with no means to extend its support. At the other end of the spectrum is *Colab* with its broadcast methods, which can be applied to an arbitrary object and, therefore provide maximum abstraction flexibility.

- **Specification flexibility.** An infrastructure should also provide a flexible specification mechanism that allows users to take full advantage of its the abstraction and sharing flexibility features at a reasonable cost. Specifically, it should support:
  - **Late specification binding.** It should be possible for the sharing modes to be specified (bound) at different times in the application's life cycle. In that respect, we distinguish between *early* binding performed at compile time by the application programmer and *late* binding performed at run time by administrators and/or end-users. Since late binding provides more flexibility, sharing infrastructures should support it unless the associated performance or specification overhead is prohibitively high.
  - **Ease of specification.** It should be easy for programmers/users to specify the exact sharing policy they need. For example, specifying a policy for a whole outline document should not require specifying a policy for each individual section. Furthermore, it should be easy for novice users to specify simple, standardized policies without precluding more complex, customized policies that more experienced users may want. In essence, this ease-of-specification requirement allows us to judge the level of automation provided with respect to the specification effort.

Suite, again, is an example of a system that fulfills both the late specification and ease of specification requirement. Since its sharing model is controlled by parameters, they can be dynamically adjusted at run time. To minimize specification effort, *Suite* uses an inheritance model, in which parameters can be specified at variable levels of granularity of the shared object's structure. It is virtually impossible to point to a system, which supports neither late binding nor easy specification, because early binding implies no specification effort. *AMF-C* [23] is a system, in which the sharing specification can be changed at any time by changing the communication links between objects, however, it's per-instance specification requires significant, even for applications of moderate size.

1.2.3 Code reuse

Often, the development of multi-user applications starts with a fully implemented single-user version of it. Therefore, the extent to which existing application functionality is reused has a direct impact on the complexity and cost of the application development. We distinguish between two types of code reuse:

- **Compiled code reuse.** In a perfect solution, the infrastructure should allow the sharing of executable single-user code, thereby eliminating development effort. However, there are fundamental, as well as practical, limitations on how much sharing flexibility can be achieved while offering compiled code reuse. Thus, if more sharing flexibility is needed, it is inevitable that the application becomes collaboration-aware. Therefore, a secondary requirement is to minimize this awareness and the corresponding development effort.
- **Incremental collaboration awareness.** To minimize the effort of making an application collaboration-aware, the infrastructure should support varying degrees of collaboration awareness. As a starting point, the infrastructure should support sharing of collaboration-transparent applications. As the application programmer incrementally modifies the application code to make it more collaboration-aware, the infrastructure should also incrementally increase the level of sharing services it provides until the application reaches the level of service required by the users. This step-wise development process ensures that the programmer invests only the minimal effort necessary to achieve the desired collaboration behavior.

Shared window systems support compiled code reuse but not incremental collaboration awareness. *GroupKit* provides a variety of mechanisms for implementing sharing—shared environments, multicast RPC, and events, and therefore, provides support for gradually increasing the collaboration awareness of the application.

1.2.4 Extensibility

Since applications are developed after the infrastructure on which they are based, it is virtually impossible for the infrastructure designer to foresee all possible future needs. Therefore, from a software engineering point of view, the infrastructure should be designed to facilitate future modifications and extensions of existing functions, as well as the integration of new ones. In the context of collaborative infrastructures, we refine this generic extensibility requirement into three specific requirements

- **Separation of shared abstractions and sharing implementation.** A sharing infrastructure should cleanly separate the shared abstraction presented to the application from its implementation. In particular, it should be possible to compose the application with a completely different sharing implementation without modifying the application code.
- **Separation of collaboration functions.** Individual sharing functions within the infrastructure should be implemented as separate modules to allow their implementations to be varied independently. For example, it should be possible to replace a lock-based concurrency control module with one based on operational transformations without disturbing the rest of the system. Furthermore, explicit dependencies among modules should be avoided to enable the transparent introduction of new functions, such as access control, without modifying the rest of the infrastructure.
- **Late component binding.** The previous two separation requirements imply that the infrastructure must also have a binding mechanism to put together a working system. Such binding can occur early (at compile time) or late (at run time). Since late binding offers more flexibility and allows the system to be reconfigured without recompilation, a sharing infrastructure should implement late binding unless the associated overhead is prohibitively high.

Component-based systems, such as the *Java*-based *JViews* [12], are best suited to satisfy this requirement because of their emphasis on fine-grained decomposition of application functions. In addition, *Java,* with its late binding and reflection mechanism significantly simplify the task of late component binding. In the other hand, *Suite,* for example was designed as a monolithic system, therefore, provides minimal support for extending/ modifying its functionality.

1.3 Thesis

The main objective of the research work described in this dissertation has been to develop new mechanisms that allowed the development of an infrastructure that better satisfies the identified generic requirements than existing systems. By better we mean that with respect to each requirement our system will perform at least as well as the best of the benchmark systems and, with respect to some requirements, it will support features not present in other systems.

We should note that the requirements discussed above are not completely independent of each other and some of them present fundamental trade offs. In such cases, our claim is that for a fixed level of support of one of the requirements, our infrastructure will be at least as good as the benchmarks. To illustrate this, consider the classification of sharing infrastructures with respect to abstraction and sharing flexibility shown on Figure 1.4 (a detailed motivation of the classification itself is presented in the next chapter).

In this case, *Suite* provides the highest relative sharing flexibility among all systems, whereas *Colab* and *GroupKit* provide the highest (possible) level of abstraction flexibility. Our contribution claim here is that, given a *Suite*-like level of sharing flexibility, our system provides a higher level of abstraction flexibility than any other systems (point G_1). Similarly, given an arbitrary object, we provide more sharing flexibility than *Colab* or *GroupKit* (point G_2). However, we do not claim that we can provide *Suite*-like sharing flexibility for an arbitrary object, which turns out to be infeasible.

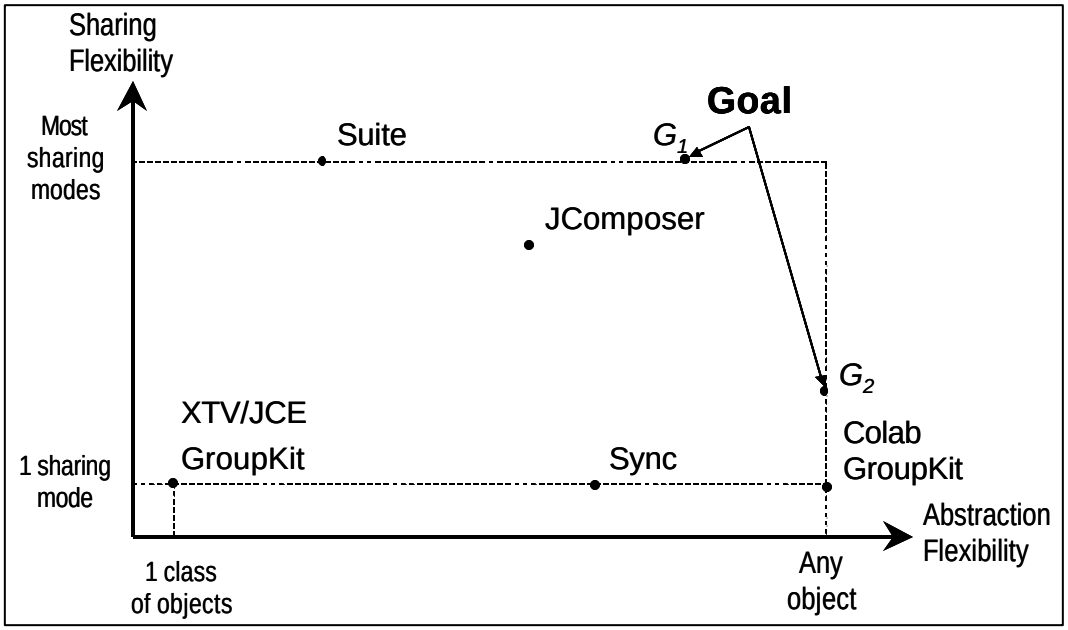


Figure 1.4 Abstraction vs. Sharing Flexibility

It is our thesis that we have developed an object-sharing infrastructure that *better* (as defined above) addresses the generic infrastructure requirements identified in this chapter than existing infrastructures. In particular, the infrastructure has achieved the following with respect to the requirements:

- *Abstraction flexibility:* for a given level of sharing flexibility, a level of abstraction flexibility at least equal to that of existing systems. Moreover, for application-based sharing it supports higher abstraction flexibility than existing infrastructures.
- *Sharing flexibility:* for a given level of abstraction flexibility, it supports a level of sharing flexibility at least equal to that of existing systems.
- *Specification flexibility:* satisfies the specification flexibility requirements at least as well as existing systems.
- *Extensibility:* it supports a component-based architecture that is more flexible than that of existing systems and provides more automation in terms of component composition.
- *Code reuse:* for given levels of abstraction and sharing flexibility, it supports at least the level of code reuse provided by other infrastructures. Moreover, for application-based sharing, it supports a higher level of both compiled code reuse and incremental collaboration awareness.

- *Automation:* it achieves the above goals at an application development cost that is comparable, or lower than corresponding solutions in other systems. In other words, given a particular sharing scenario in which the above requirements are satisfied to a certain degree, the development effort involved using our infrastructure will not exceed that of analogous solutions.

In order to overcome some of the limitations of existing approaches without compromising their achievements, we have developed a set of new mechanisms, upon which our infrastructure is based.

Specifically, we formalize the notion of a *programming pattern* and provide a mechanism that allows its incorporation into the application development process. A programming pattern is a generalization of the notion of programming interface used in object-oriented languages. It is logically equivalent to a *family* of interfaces, and allows sharing, as well as other infrastructure services, to be implemented on a per-pattern basis. A programming pattern is defined *implicitly* by the application developer through the consistent use of programming naming conventions. Such patterns are in widespread use nowadays, because they are considered a part of good programming practices. Nevertheless, they have remained largely informal and unused in the development process. To take advantage of these patterns, we define an *XML*-based specification language that enables programmers to concisely describe them. Based on the pattern specifications, the infrastructure can decompose an object into a set of logically autonomous subcomponents, or *properties*, and, correspondingly, break down the problem of sharing the entire object into sharing of its properties. This process is then recursively applied to the values of the properties, until terminal nodes in the object structures are reached.

This approach allows the infrastructure to provide sharing of all the abstractions provided by existing systems at a comparable cost. In addition, it can provide sharing of an extensible class of programmer-defined abstractions at no additional cost, whereas a comparable solution for existing systems would require effort potentially proportional to the number of new programmer-defined abstractions.

Another contribution is the development of new generic infrastructure services, such as object diff-ing and *XML* object serialization, which improve sharing flexibility. Specifically, our pattern-based allows these to be implemented at a very local cost compared to the current

alternative of object-specific manual implementations. Thus, while the general idea of diff-ing, for example, has been employed by a number of systems, we are unaware of any other sharing infrastructure that has attempted to provide a generic solution for object diff-ing.

We also refine and formalize the layer-based application model presented above by introducing an *XML*-based layer specification language. Application layer specifications allow the infrastructure to correctly separate sharing of semantic and UI objects without any adding or modifying any code and enables the dynamic changes in the sharing architecture in a more general setting then currently possible.

Architecturally, our infrastructure design is based on the idea of *property handlers*, which encapsulate the property-specific code related to the implementation of an infrastructure functions. The binding of handlers to properties is given in the pattern specifications and can, therefore, be changed easily without modification to the application code, or the rest of the infrastructure. At the same, this loose binding is sufficient to make the composition of shared objects and sharing functions automatic.

Our component-based architecture also greatly facilitates satisfying the code reuse requirements. With respect to compiled code reuse, the specification-based binding of objects and services allows the infrastructure to use *Java*'s computational reflection capabilities to compose compiled application code with the infrastructure. By the same token, infrastructure services can be added or modified by adding/modifying the handler implementations and the whole process would be transparent to both the application and to other services in the infrastructure. Hence, the infrastructure provides a mechanism to incrementally increase the application's collaboration awareness through the incremental addition of handler code. Furthermore, since programming patterns are predominantly a function of programming style and depend very little on the application, they can readily be reused across applications.

The primary tool in evaluating our infrastructure design and implementation is a comparison with existing infrastructures. For that purpose, we survey in detail a number of influential collaborative infrastructures and identify benchmark systems, which provide the best solution with respect to one, or more, of the requirements. We then compare the benchmarks with our own work using the above outlined criteria. The comparison consists of two parts—in the first part, we compare our conceptual design with that of the benchmark systems and present a basic argument explaining exactly how some of the shortcomings of

the benchmarks are overcome. The second part consists of comparing the implementation of a number of test cases using our infrastructure and comparing the results with those of the benchmarks.

1.4 Organization

The dissertation is organized as follows. In this chapter we already motivated our research and presented a comprehensive set of requirements for sharing infrastructures. In Chapter 2, we review related work, which includes a number of collaborative infrastructures, which we use later as benchmarks for evaluating our own. In Chapter 3 we describe the conceptual framework behind our infrastructure—programming patterns, property handlers, application layer descriptions—and relate it to that of other systems. In Chapter 4 we lay out the details of the implementation with the specific issues and approaches that arise in the process. In Chapter 5, we describe our experience both in building new collaborative applications using our infrastructure and in converting existing single-user applications into multi-user ones. We also share our experience in using some of the developed mechanisms for non-collaboration purposes. In Chapter 6 we evaluate our infrastructure with respect to the requirements using the criteria presented above and compare them to the benchmark systems. In Chapter 7 we draw conclusions from our work and outline directions for future applications and extensions of this research.

2. RELATED WORK

In the previous chapter, we discussed a comprehensive set of requirements for collaborative infrastructures. In this chapter, we outline the design and implementation choices made in existing infrastructures that have addressed at least some of these requirements. Following the basic three-layer decomposition introduced in the introduction—user interface, core application and repository—we discuss the advantages and drawbacks of implementing sharing of each of them, by examining the design of representative system. We also use this discussion to show in detail the rationale behind our choice of benchmark systems, which we use in our evaluation process in Chapter 6.

2.1 User Interface Sharing Infrastructures

Sharing of the UI layer is one of the most popular approaches to building groupware systems and has significant advantages to its credit. From a user's point of view, sharing of the UI represents a simple and intuitive solution, which requires virtually no new skills from the participants. From a developer's point of view, UI sharing offers significant automation and code reuse, as well as the appeal of cleanly separating the design and implementation concerns of the core application functions from those regarding the collaboration. On the downside, however, users may find the UI sharing model too restrictive, as it is inherently unsuitable for a whole spectrum of collaboration scenarios.

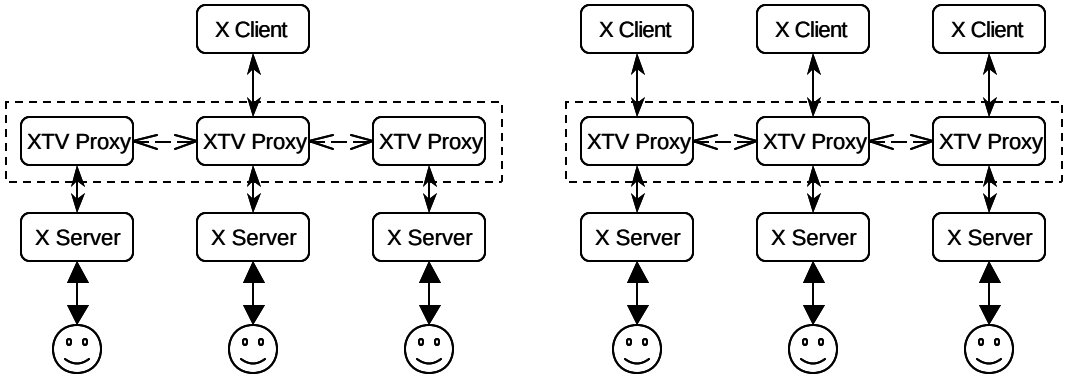
Based on their choice of a basic shared abstraction, we subdivide UI sharing infrastructures into three categories—shared window systems, shared UI toolkits, and shared screen system—and describe their most important characteristics with regard to our requirements.

2.1.1 Shared Window Systems

Shared window systems, such as *XTV* [1], are based on the notion of a *shared window*. A shared window has multiple replicas—one for each participant—that are synchronously shared. That is, all users are presented with the same up-to-date view of the application

window and the application can take input from any one of the replicas. Thus, by sharing all application windows, the infrastructure achieves the WYSIWIS sharing described in the previous chapter (Figure 1.1).

Conceptually, the implementation of a shared window system relies on intercepting and properly distributing the communication streams between the application and the window system. In XTV, for example, this is achieved by installing proxies, which mediate the communication between the X servers running on each user's machine and the one (or more) X clients where the application (replicas) run. As illustrated by Figure 2.1, *XTV* has two versions—a centralized one where all X servers are connected to a single X client, and a fully replicated one where each server is paired with a local replica of the client.



a) Centralized *XTV* b) Replicated *XTV*

Figure 2.1 XTV Architecture

Regardless of the configuration, *XTV* (as well as other shared window systems) must maintain two properties at all times for the system to work properly: 2

- All application replicas should receive the same ordered stream of user input events. Since applications are deterministic, this in turn ensures that the stream of output events generated by application replicas will also be the same[1].
- All window servers should receive the same ordered stream of application output events. Furthermore, since the window servers may handle events asynchronously, the infrastructure must ensure that the events are delivered to the servers at the

[1] Non-deterministic applications, as well as operations with side effects, such as printing, present special cases, which are beyond the scope of this discussion.

appropriate rate to avoid, for example, an attempt to change the title of a window that is yet to be created [4].

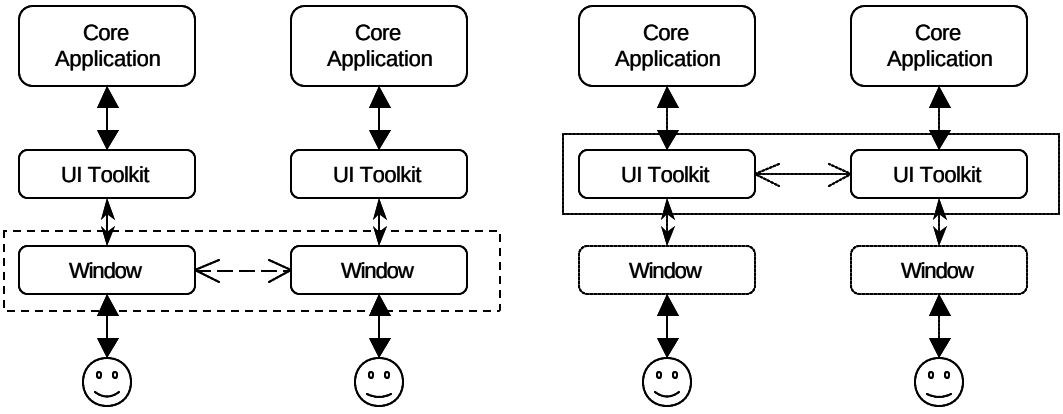
Since the outlined interception of the event streams can be implemented transparently to both the application and the window system, this approach allows virtually any application written for the supported window system to be shared at no additional development cost. In terms of our requirements, shared window systems provide a maximum level of automation and code reuse and, therefore, a useful benchmark in evaluating our infrastructure.

One of the main disadvantages of shared window systems, however, is that the automation and reuse features come at the expense of sharing flexibility. Namely, only a single mode of sharing—WYSIWIS—is supported. Some researchers have proposed approaches that address this issue in a limited context. For example, Chung's log-based approach [4] allows a shared window system to be extended to possibly support some forms of relaxed-WYSIWIS sharing. However, it does not address the fundamental limitations of shared window systems, which stem from the fact that the sharing of the window-based user interface of an object is used as a means of sharing the object itself. Since the two cannot be separated, it is not possible to implement any form of semantic sharing.

By the same token, a shared window system cannot allow concurrent access to the shared object because it cannot distinguish concurrent user actions that are in conflict from the ones that are not. Suppose two users decide to edit two different text fields—depending on the application semantics, this may, or may not be allowable, but the sharing infrastructure has no basis to make the right decision. In particular, the structure of the UI presentation is not a reliable proxy for the logical structure of the underlying semantic object because structural dependencies among UI objects are often dictated by formatting needs. Furthermore, if concurrent updates lead to divergent replicas, the infrastructure has no means to restore consistency to the underlying semantic object. Thus, shared window systems take a conservative approach and limit the number of active users to one. More generally, a decision based solely on the observed structure, even if it is that of the semantic object, does not guarantee correctness. However, the higher the level of abstraction at which the decision is made, the better the chance that it is, in fact, correct.

2.1.2 Shared UI Toolkit Systems

Another approach to sharing the user interface, represented by systems such as *JCE* [2] and *Habanero* [3], is based on creation of shared UI toolkits. With the recent emergence of *Java* as a mainstream programming language, this approach has gained in popularity because *Java* provides a standard *Abstract Window Toolkit* (*AWT/Swing*) library, which can be extended to provide sharing. The *AWT/Swing* toolkit provides a more abstract view of window systems and, therefore, a *Java* application's user interface is no longer bound to the particular implementation of the window system but to the *AWT* API. Consequently, the application can run on any *Java* platform without modification. Conceptually, this means that the toolkit adds a separate layer of abstraction and we can break down our original UI layer into window and toolkit layers. Consequently, we can represent shared window and shared toolkit systems as providing sharing at the window level and toolkit level, respectively (Figure 2.2).



a) Shared window system b) Shared UI Toolkit system

Figure 2.2 UI Sharing: Shared window and shared toolkit architectures

Since UI toolkits seek primarily to provide a common interface for different window systems, they provide relatively few abstractions of their own. Hence, the differences between sharing at the window and toolkit layers are subtle and, by and large, inconsequential to the end user. As an example, consider a collaborative session of our shared outline with one of the users is running an application replica under X Windows, whereas the other one is running it under MS Windows. The two users see the same window content in terms of text fields, button, scrollbars, etc, but may not see the same images pixel-

for-pixel because of platform-specific differences in default fonts, colors, menu appearances, button shapes, etc.

Nevertheless, the characteristics of the collaboration support provided by sharing the window system and by sharing the UI toolkit are very similar:

- Full automation and reuse for all code based on the toolkit;
- Support only for synchronous collaboration;
- No support for concurrent user updates of the shared object. As already discussed in the context of shared window systems, the infrastructure cannot make an informed judgment call as to what concurrent user updates are permissible and, therefore, has little choice but to prevent concurrent access. The argument applies to shared toolkits because a toolkit does not create new abstractions but simply hides the native implementation details of the UI components.

2.1.3 Shared Screen Systems

Another alternative to shared windows system is based on lowering the level of abstraction at which the sharing takes place. Infrastructures, such as VNC [19], have chosen to implement sharing of the pixel image seen by users by sharing the contents of the frame buffer of the user machine. Since at the pixel level, the sharing infrastructure cannot distinguish among application windows, the whole screen image must be shared and the WYSIWIS model is taken to its extreme by essentially projecting the image of one user's screen onto another's.

With respect to our requirements, VNC-like systems have virtually the same characteristics as the other UI-based sharing system—full automation and reuse, no sharing flexibility, and no concurrent access. The only difference is in the range of supported applications—because of its pixel-level of abstraction, these systems can share just about any application on any supported platform. However, this low-level of abstraction also makes the implementation highly dependent on the graphics hardware.

2.1.4 Summary

In summary, we presented three separate approaches in sharing infrastructures that apply the same idea of sharing the user interface of an object as a means of sharing the object itself.

The only conceptual difference among the three is that their sharing mechanisms are designed for sharing different layers of the user interface (Figure 2.3).

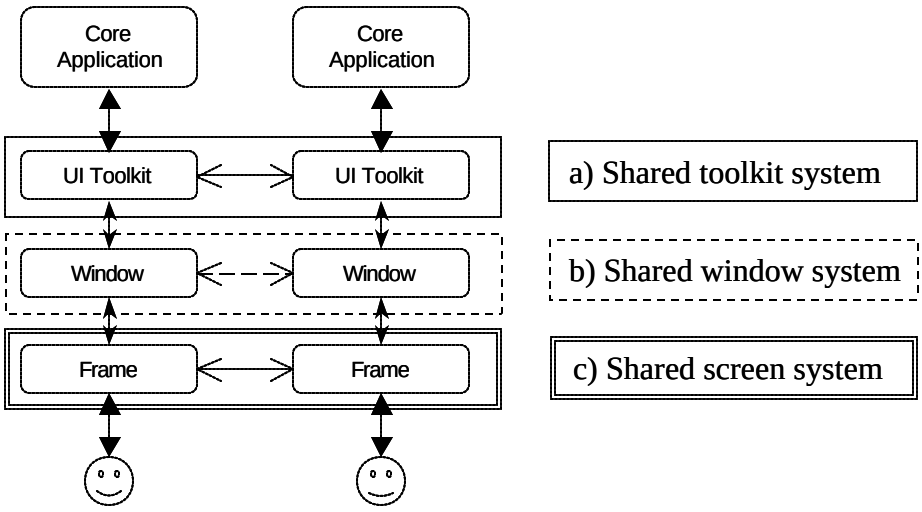


Figure 2.3 Shared UI Architectures

We argued that, regardless of the merits of each individual approach, we can treat them all as representing a single class of systems that share common characteristics with respect to our infrastructure requirements. Namely, UI sharing systems:

- allow the automatic sharing and reuse of compiled code,
- limit sharing flexibility by supporting only WYSIWIS collaboration, and
- limit sharing flexibility by allowing only one user at a time to interact with the application.

Thus, it is sufficient for the purposes of our evaluation to choose one representative system, with respect to which we can compare our results. Given that our infrastructure implementation is *Java*-based, we have chosen another *Java*-based system—*JCE*—as our benchmark system. Although, conceptually, *Java* is not essential for our evaluation, it allows us to provide an apples-to-apples *implementation* comparison.

2.2 Application-Based Sharing Infrastructures

So far, our discussion shows that sharing based on a shared UI abstraction places inherent limitations on the sharing flexibility the infrastructure can support. Therefore, to avoid such limitations, the infrastructure must provide shared abstractions that allow the developer to

implement application-based sharing and, thus, separate the sharing of the (user) *view* of an object from the sharing of its data representation, or *model*.

2.2.1 GroupKit

One of the early and successful systems that provided application sharing was *GroupKit* [20]. One of the design goals of *GroupKit* was precisely to provide support for collaborative extensions of application design models that separate the semantic object from its UI appearance, such as the Model-View-Controller (MVC) [15] and Abstraction-Link-View (ALV) [14] paradigms.

To directly support model sharing, *GroupKit* offers *shared environments* (dictionary-style data structures containing keys and associated values) whose replicas, once instantiated, are automatically kept consistent by the system. Furthermore, environments provide a callback notification mechanism through which applications can learn of operations performed on the shared environment—insertions, deletions, and replacements. Thus, *GroupKit*'s environments can be used to implement a shared version of the data representation of the object, and the callback mechanism can be used to update the local view of each user. This is illustrated by Figure 2.4, where the arrows show the flow of events initiated by user *A*'s action. First, the local controller receives the user action and transforms it into an operation on the local model. Next, the model updates the local view and the infrastructure automatically applies the update to the model replica user *B*'s machine. Finally, user *B*'s model updates its local view object.

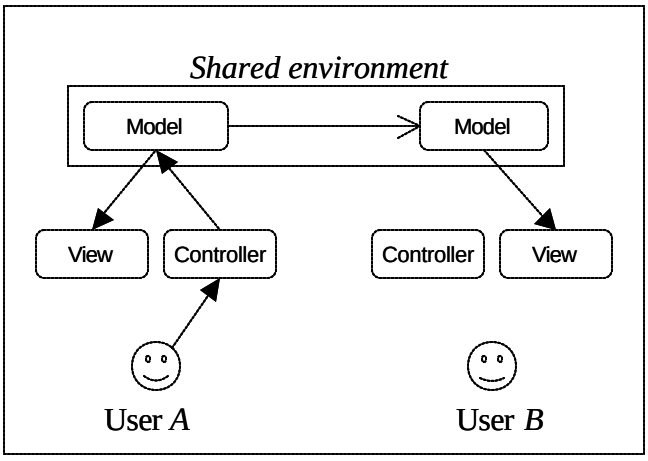


Figure 2.4 Collaborative extension of the MVC architecture using *shared environments*

The main difference of this model, compared to UI sharing, is that the decision of when to update the UI is not bound by the infrastructures. For example, if the callback immediately updates the local display in response to every remote operation being applied to the local replica of the shared environment, this would result in WYSIWIS sharing. At the same time, the callback procedure has the flexibility to arbitrarily delay refreshing the UI until some other criterion is met, e.g., the user presses a `<refresh>` button. However, *GroupKit*'s mechanisms were never intended for such use and, in all likelihood, were never actually employed in this manner.

Clearly, the above design offers a very flexible solution in choosing the level of sharing for the UI. However, this flexibility comes at the added cost of developing application-specific callback routines that implement it. Furthermore, the developer must have a good understanding of the distributed issues involved to avoid potential problems, such as deadlocks, and cyclic event notifications.

Another point we would like to make is that, while the UI can be shared flexibly, *GroupKit* offers no sharing flexibility with respect to the shared environment—they are always shared synchronously. Hence, while users may see different renderings of the shared object, the object itself cannot have separate versions for some user-controlled amount of time.

The choice of a single shared abstraction—environments—presents another important limitation. To illustrate this, consider the implementation of our example outline object in *GroupKit*. As it turns out, environments are inherently ill-suited for this task because they are not designed to deal with sequences and recursive structures—in our case an outline has a sequence of sections and each section can have subsections several levels deep. Therefore, we would have to implement our own shared structures to support the outline application using lower level communication mechanisms, such as *multicast RPC*. Multicast RPC is *GroupKit's* version of the idea of simultaneously performing remote procedure calls on multiple hosts. The original idea, known as *broadcast methods*, was conceived and implemented in *Colab*. The main difference is that, in *GroupKit*, there exists the option of *asynchronous* multicast RPC, which does not block the sending process until delivery is complete. This provides more flexibility but also may incur non-trivial programming overhead as the application will need to supply a coordination mechanism.

2.2.2 Colab

In *Colab, broadcast methods* [22] provide a mechanism that automatically and synchronously replicates a local method invocation on a set of distributed object replicas. The rationale behind broadcast methods is to take a single-user implementation and turn it into a multi-user one by identifying and marking all methods modifying the shared object's state as *broadcast*. In other words, our shared outline would look something like:

```
public class Outline {
  String  getTitle();
  broadcast void setTitle(String title);

  Section getSection(int i);
  int     getSectionCount();
  broadcast void setSection(int i,Section s);
  broadcast void insertSection(int i,Section s);
  broadcast void removeSection(int i);
}
```

Assuming that all object replicas start with the same state, broadcast methods would keep them consistent by automatically propagating and remotely applying changes made to any one of the replicas. This simple mechanism is fairly generic and can potentially support the sharing of almost any object[2]. Therefore, *Colab* supports the highest level of abstraction flexibility of all sharing infrastructures, as well as a very high level of code reuse.

Unfortunately, the price for this abstraction flexibility is the complete lack of sharing flexibility—objects can only be shared synchronously and the decision which objects are shared is bound at compile time. This leads us to the conclusion that *Colab* does not fulfill our requirements in terms of specification flexibility and extensibility.

2.2.3 Suite

The design of another infrastructure, *Suite* [8], was focused on providing a high level of abstraction, sharing, and specification flexibility at minimal development cost. Conceptually, the system is based on the idea that every interactive application can be represented as an editor through which the user edits a specific data structure. In *Suite*'s case, the infrastructure supports the automatic sharing of arbitrary programmer-defined data types in *C*.

To achieve this, *Suite* uses the type and variable declarations provided by the programmer to extract the structure of the shared data and to generate a corresponding user interface that

[2] Some methods cannot be declared broadcast, such as the ones invoking other broadcast methods (directly or indirectly).

allows the collaborative editing of the shared data. In other words, *Suite* also adheres to the model-view application architecture, with the view layer being automatically generated and maintained by *dialog managers*. Thus, based on the following *C* declarations of the example outline, *Suite* can automatically provide the user interface and sharing functions needed for its collaborative editing.

```
typedef char * String;
typedef struct {
   unsigned num_sections;
   Section *sections;
} SectionSequence;
typedef struct {
  String heading;
  SectionSequence subsections;
} Section;
```

```
typedef struct {
  String heading;
  SectionSequence *subsections;
} Section
```

Central to Suite's sharing model are the notions of *active variable* and *interaction variable*. An active variable is an application variable that is displayed to and can be edited by the user through an interaction variable. An interaction variable is a user's local version of an active variable, which is created when the user connects to the object. Thus, to allow users to share an active variable, Suite creates a corresponding interaction variable for each user and connects them to the shared active variable (Figure 2.5).

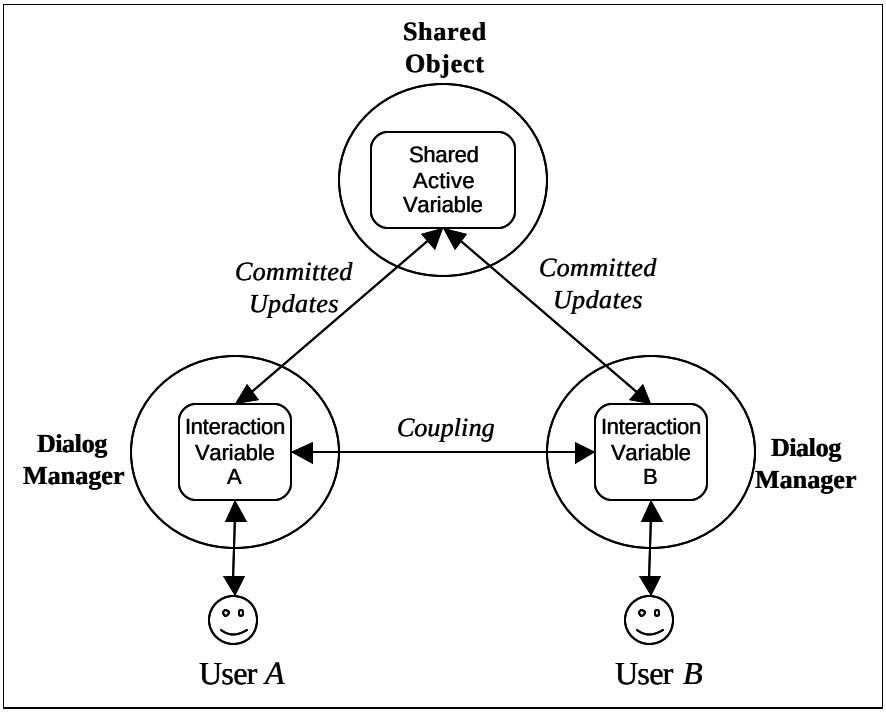


Figure 2.5 *Suite*'s sharing model

Users work on their local versions and must explicitly commit their changes in order to modify the shared variable. Committed updates are always communicated to all participants. In addition, there is a coupling mechanism, which allows users to see each other's work at various stages of its completion. This is controlled by two parameters: a transmission parameter, which sets a condition for exchanging updates based on the communication operation being performed on the variable, and a correctness parameter which controls the exchange based on the level of correctness of the variable value. The basic rule for transmitting changes is as follows: the value of an updated interaction variable is transmitted upon execution of any communication operation on the variable that is greater than, or equal to, the value of its transmission parameter if the correctness of the value is at least as high as the one specified by the correctness parameter.

The values of the transmission parameter reflect the following ordered set of communication operations defined by the system: *Increment, Complete, TPeriod, TTime,* and *Transmit*. By default, every variable update, such as inserting a character, is an *Increment* operation. A *Complete* operation is executed whenever the user has indicated that he is finished editing the value, e.g., by hitting `<tab>` or `<return>`. *TPeriod* and *TTime* refer to a

periodic operation (e.g., every 30 min) and a scheduled time-of-day operation (e.g., at 5pm), respectively. The *Transmit* operation is executed whenever the user explicitly requests it by pressing a `<transmit>` button.

The possible values for the correctness parameter (in increasing order) are *Raw*, *Parsed*, *Validated*, and *Committed*. By default, any variable value is *Raw*, unless it has undergone a successful syntactic check (e.g., the entered string must be a 5-digit number) after which it is elevated to *Parsed*. If the value has also passed a check for semantic correctness (e.g., the 5-digit number must be a valid ZIP code), it becomes *Validated*. As already suggested, *Committed* values are explicitly designated by the user by executing a *commit* command (e.g., pressing a `<commit>` button).

The coupling parameters for sending and receiving updates do not have to be identical. Thus, user *A* may decide to make available all of his updates to everyone by setting the transmission threshold to <*Increment*, *Raw*>, however, user *B* may decide that this is too much of a distraction and may choose to receive only *Committed* updates. Going a step further, *Suite* allows users to have potentially different coupling parameters with each participating user, thereby enabling all possible combinations of user subgroups and sharing modes to be supported.

Another important abstraction in the *Suite* model is the notion of a *value group*, which groups together a set of related interaction variables and stores sharing, as well as other, attributes common to these variables. An inheritance relation that follows the ancestral relationships within the shared data structures is defined among value groups to allow more specific value groups to inherit attributes from more general ones [9]. For example, a coupling parameter specified for the outline document provides a default for all sections and subsections in the object. However, a parameter specified for section 1, for instance, overrides this default for all of its subsections and their dependents. Thus, to find the coupling parameters, say, for the heading of section 2.1, *Suite* would first look for a specific value given by the user for that heading. If none is found, the system would lookup the parameter for the value group associated with the structural parent of the heading—section 2.1, and would successively lookup the parameters for section 2, and the root outline object, if necessary.

An alternative approach to parameter specification in *Suite* is based on type-instance relationships. For example, one could provide a default specification for all instances of type Section and might expect to see the heading of all sections (including 2.1) to be shared according to this parameter. Thus, both of the approaches have useful applications in actual usage scenarios. However, they also create ambiguity as to which hierarchy—the structure-based or the type-based—should be traversed to find defaults. Therefore, *Suite* maintains a separate attribute, which tips the systems to the desired method of resolving this conflict.

For each shared entity, such as a section, users can choose between *value* or *view* coupling. Value coupling specifies sharing of the data representation of the shared entity whereas view coupling specifies sharing of its user interface. Furthermore, coupling of other UI state—scrollbars, pointers, window sizes/positions—can also be independently specified. Thus, if all aspects of the UI presentation are shared, this effectively leads to WYSIWIS sharing. However, if only value coupling is chosen, then we get semantic sharing, which allows users to have separate views of the object while maintaining semantic consistency.

To summarize, our review of *Suite* shows that it is the infrastructure that best satisfies our sharing and specification flexibility requirements. In particular, its exhaustive parameterized coupling model offers *automatically* more sharing modes than any other sharing infrastructure that we are aware of. We draw a similar conclusion about its comprehensive sharing specification model—it satisfies all aspects of our specification flexibility requirement better than any comparable system. Therefore, we have chosen *Suite* as our comparison benchmark with respect to sharing and specification flexibility.

Suite's primary limitations are with respect to abstraction flexibility, which is constrained by two factors. The first one is that the system is designed to share a fixed set of concrete data types and, therefore, its approach cannot be directly applied to abstract data types, such as *Java* classes with no public variables. The second one is that the whole coupling mechanism is tied to the generated user interface, receiving from it UI-specific change notifications and invoking UI-specific operations to update the UI state. Therefore, *Suite* cannot provide collaboration functions for an application with a graphical, or custom-built user interface.

With respect to code use, *Suite* does not offer compiled code reuse and does not support incremental collaboration awareness. From an architectural point of view, *Suite* is a monolithic system and, therefore, does not satisfy our extensibility requirement.

2.2.4 JViews

JViews [12] is another infrastructure designed to support the sharing of applications based on the editing model. However, unlike *Suite*, its solution advocates a component-based architecture that allows applications with custom-built user interfaces to be shared. Since *JViews* does not generate the user interface, and, hence, does not control the flow of updates, it must rely on cooperation from the application. Shared objects must implement a two-phase system-defined event protocol—they must preannounce intended changes (and be prepared to roll them back), as well as announce implemented changes after the fact by issuing *change descriptions* (events).

The infrastructure implements application sharing by logging, distributing, and replaying the state changes according to the current, user-selected sharing policy. *JViews* supports asynchronous and synchronous sharing, as well as user-mediated sharing. The latter means that changes from remote users are displayed in a list, and the receiver can pick the changes he wants to see applied to his local replica.

Unlike *Suite*, *JViews* does not allow policies to be specified on a per object basis and does not distinguish between sharing of the object and its view. This is due to the fact that changes affecting only the view presentation, such as moving/resizing a window, are mixed in with semantic changes that affect the shared artifact, such as text editing, and cannot be controlled separately. Therefore, if a user wants semantic sharing, he must preview and selectively apply only the semantic changes—a manual process that can lead to mistakes and consistency problems. Furthermore, if all changes to both the model and view layer are eventually communicated regardless of whether there are needed, then situations in which bandwidth is an issue would not be adequately supported.

With respect to abstraction flexibility, *JViews* supports the sharing of an extensible set of programmer-defined abstractions that implement system-defined interfaces. However, it does not directly support the sharing of recursive structures—in order to achieve such sharing, each component in the structure must be explicitly connected to the infrastructure.

On balance, the best features of *JViews* stem from its component-based approach, which is naturally suited to support extensibility and code reuse by allowing components to be added or substituted without affecting the functions of rest of the infrastructure. Also, incrementally increasing the application's collaborative features can readily be accommodated. Therefore, we use *JViews* as our benchmark in extensibility and code reuse for application-based sharing.

2.2.5 DISCIPLE

DISCIPLE [Wang, 1999 #20] is another *Java*-based sharing infrastructure, whose design approach is based on components. It is designed to share arbitrary *JavaBeans* objects by taking advantage of their standardized *property* and event model. A (*Java*) bean is a regular object, which follows a prescribed method naming convention and communicates through a standardized events. The naming conventions allow an external agent—the *Java Introspector*—to discover different attributes, or *properties*, of the object. A property is defined by a pair of 'getter' and 'setter' methods, such as the *getTitle/setTitle* methods used in our outline example:

```
public class Outline {
  String getTitle();
  void setTitle(String title);

  Section getSection(int i);
  int     getSectionCount();
  void setSection(int i,Section s);
  void insertSection(int i,Section s);
  void removeSection(int i);
}
```

Thus, based on the *JavaBeans* conventions, the *Outline* object above has a property called *title*. In addition, a proper bean object would also allow interested objects to register as listeners and be notified whenever a property value changes.

Based on this model, *DISCIPLE* is able to provide sharing of any bean object by registering the infrastructure as a listener and propagating and applying the changes to remote replicas. While this is very similar to what *JViews* does, the main conceptual difference is that *DISCIPLE* distinguishes between model and view objects and allows those to be shared separately. At the same time, it only implements synchronous sharing, thereby limiting sharing flexibility.

From the point of view of abstraction flexibility, bean objects represent an interesting class of objects: on the one hand they allow systems, such as *DISCIPLE*, to automatically

support the sharing of an extensible class of programmer-defined types; on the other hand the bean model is too restrictive to cover many objects with relatively simple logical structure. For example, our outline has only one property—title—and that is clearly insufficient to fully describe its structure. Overall, beans represent an interesting class of objects, disjoint from what most other object-sharing infrastructures, such as *Sync* (described in the next section) support. Therefore, we use *DISCIPLE* as one of our benchmarks with respect to abstraction flexibility.

With respect to reuse and extensibility, these are well-supported by the underlying *JavaBeans* framework and, hence, well-supported by *DISCIPLE*.

2.2.6 AMF-C

Another component-based infrastructure, *AMF-C* [23], has taken on the issue of providing sharing flexibility by using both the structure of the shared objects (like *Suite*) and an event-based communication protocol (like *JViews*). In *AMF-C*, an object can be shared automatically if it publicly advertises its logical structure through the use of *facets.* A facet is responsible for a particular aspect of the object's behavior, such as *presentation* (to the user), *abstraction* (functional kernel), and (interaction) *control*, in the PAC model [6]. Facets have communication ports through which objects interact by exchanging messages. The event flow between objects is specified using a graphical formalism similar to the standard notation for circuit design. The language allows the flow to be controlled using *administrator* components (e.g., lock administrator) that can stop event propagation, as well as basic logical functions– *AND*, *OR*, etc.

As an illustration, consider the implementation of synchronous sharing for the outline example using *AMF-C*. First, the outline object must define the standard presentation, abstraction, and control facets as well as a *distant* facet—a facet responsible for replaying actions received from remote sites. Then, the sharing logic must be specified by connecting the relevant communication ports—in the simplest case, just connecting the output communication port of the abstraction facet to the remote replay facet achieves a sharing mode similar that of broadcast methods. Considering the fact that the graphical language allows arbitrary connections to be made between components, *AMF-C* provides a mechanism

to specify and implement a wide variety of sharing modes. However, this comes at a relatively high development overhead, which comes from:

- Low-level event processing—the programmer must specify the sharing logic using low-level events. Consequently, the shared object must implement a messaging protocol for communicating and applying object updates, as well as methods for manipulating facets and ports.
- Cumbersome sharing specifications—the specification process itself is tedious because sharing specifications are made on a per-object basis. Furthermore, the graphical formalism chosen would lead to complicated diagrams even for applications of relatively modest complexity.

Overall, *AMF-C* presents an interesting approach, which offers the potential for significant specification flexibility, however, its emphasis on low-level primitives means that the provided level of automation is significantly lower than that of other systems, such *GroupKit* and *Suite*.

2.2.7 Summary

In summary, application-based sharing has the potential to provide the highest level of flexibility as it permits any aspect of the application to be shared. Our survey shows that infrastructures supporting application-based sharing provide developers with reusable mechanisms that fall into two basic categories—group communication services and shared programming abstractions.

Group communication services, such as *Colab*'s broadcast methods, provide a lower level of abstraction that makes them applicable in a general context. Ultimately, however, what the programmer actually needs are shared programming abstractions that fit the needs of the specific application they are developing. In that sense, group communication is necessary but not sufficient to fulfill the needs of the programmer. Also, any limitations in the communication service can affect the capabilities of the shared abstractions built on top of them. For example, shared objects based on broadcast methods are inherently limited to synchronous sharing.

Shared programming abstractions, such as *GroupKit*'s shared environments, generally provide high automation because they can be directly used in the application development.

The main challenges for the infrastructure are to provide enough abstraction flexibility to accommodate the abstractions of a large class of applications, as well as sharing and specification flexibility to allow a wide range sharing modes.

The main goal of our work is to provide higher abstraction flexibility for application-based sharing than existing systems and provide sharing and specification flexibility at least equal to the best infrastructure in this class.

2.3 Repository-Based Sharing Infrastructures

2.3.1 Traditional Distributed File Systems

Traditional distributed file system (DFS), such as *AFS* and *NFS*, implement sharing by giving users remote access to the same (logical) file over the network. In reality, users may be accessing different physical replicas of the file, in which case the system enforces strong consistency among the replicas to ensure that the single-copy abstraction is not broken.

The obvious advantages of DFS are that the sharing mechanism can be composed with any application that uses the file system. Thus, DFS provide a similar level of automation and reuse as shared UI systems. Another similarity is that they both serialize user access to the shared object and do not permit concurrent updates. This is the result of the fact that in both cases the sharing infrastructure has no knowledge of the application and must act conservatively in order to guarantee consistency.

Overall, providing protection boundaries between users is a much higher priority for DFS than facilitating collaboration. Therefore, classic distributed file systems do not provide a mechanism that allows applications to extend the basic file-level sharing to a finer granularity, or to modify the file sharing semantics.

2.3.2 Coda

Coda [17] is a distributed file system specifically designed to allow concurrent file updates in the presence of disconnected and weakly connected participants. Whenever disconnected, users work on a cached local copy of the file and the system keeps track of all operations performed on the file. Upon reconnection to a server, *Coda reintegrates* the logged operations with the operations performed on the master file during disconnection. A variation of this process, called *trickle reintegration*, can be performed as a background

operation to support weakly connected clients using narrow band channels of communication.

With respect to our requirements, the main feature of *Coda* is its automatic and application-transparent support for file-based sharing. Like traditional DFS, this model works reasonably well for applications that organize their data in multiple files, because there is less chance of users modifying the same file and having a conflict. However, for applications, such as word processors that use one big structured file, the file level granularity is inadequate. Therefore, *Coda* allows applications to supply specialized procedures for automatic conflict resolution. These routines understand the underlying structure of the stored data and can precisely determine whether two concurrent updates are in conflict or they can be reconciled without requiring user intervention.

From the point of view of abstraction flexibility, applications are given a mechanism through which they can extend the list of supported shared abstractions beyond the basic file system abstractions. The basic problem, however, is that any flexibility gains come at the expense of the application developer who must design and implement the merge procedures. From the point of view of extensibility, though, *Coda* supports the basic idea of offering different level of service to applications with different levels of collaboration awareness, albeit not in an incremental fashion.

2.3.3 Lotus Notes

Unlike the repository systems discussed so far, which use files as the basic unit of sharing, *Lotus Notes* [16] provides sharing based on the notion of *document databases*. A document database is a flat collection of *Notes* documents, which in turn are records of arbitrary objects. Databases may also contain *forms*, responsible for entering and displaying data, and *views*, which show summaries of database content. Programmers can use *formulas* and *macros* to access and manipulate objects in the database. A formula is a composition of *Notes @ functions* through which documents are accessed, whereas a macro is a formula that performs a procedure on the database. All of the described objects can be shared through a process of successive replication and merging.

Notes does not use the typical notion of sharing, which is based on the idea of eventual consistency—if there are no active transactions pending, then all replica must converge to the

same values (network conditions permitting). *Notes*, however, considers replicas to be consistent if they have equivalent document versions. In other words, not all replicas will necessarily have the same set of documents.

The sharing process is controlled by the chosen replication topology and four database attributes. The possible topology configurations include peer-to-peer, hub-and-spoke, tree, as well as several other variants of these. The four attributes include *access control lists, read access lists, replication settings,* and *replication formulas*. Access control lists determine for each replica which database elements may be exchanged with other replicas. Read access lists are defined on a per document basis to control who may receive them during the merge process. Replication settings give finer control over the replication process, such as the ability to exclude deletions of document from being replicated. Replication formulas allow developers to specify criteria which select the documents to be shared.

Notes' semi-structured databases provide higher level of abstraction and sharing flexibility than file systems. Concurrent updates are considered in conflict only if they are made to the same document, however, the resolution must be handled manually by the users. Also users have a lot of flexibility in controlling which documents should be shared but have no control over the sharing semantics except for the manual reconciliation.

2.3.4 Bayou

Bayou [24] is another sharing infrastructure that supports disconnected operations. Like *Notes*, it is a database-oriented system, which defines *tuples* (similar *Notes'* documents) as its basic unit of sharing. The main difference is that *Bayou* specifically addresses the need for greater sharing flexibility by providing more flexible mechanisms for conflict detection and resolution. Specifically, *Bayou* provides a scripting language that allows each individual write to the database to be accompanied by a conflict-detection query and conflict resolution procedure. If the actual query result defers from the expected, this is flagged as a conflict and an application-specified procedure is invoked to resolve it.

Thus, *Bayou* provides more sharing flexibility than *Notes*, although taking advantage of it may requires writing lower-level code, which lowers automation and increases the complexity of the sharing specification. Another problem is the relatively low abstraction flexibility provided by the system. For example, casting the recursive data structure of our

outline document would not be a trivial task and will require an additional application layer to be built to support the translation between the application objects and database records.

2.3.5 TACT

TACT [25] is another infrastructure, which, like *Bayou*, implements a database-centered sharing model and has similar design features with respect to our requirements. TACT's main advantage over *Bayou* is that the sharing is controlled through the specification of three parameters—*numerical error*, *order error*, and *staleness*. Each of these parameters provides some measure of the differences between replicas. Whenever any of the observed values of these parameters exceeds an application-specified threshold, an automatic exchange is triggered to bring the replicas back in line with the specification. For example, if the specification is set to `(0,0,0)`, then no differences are allowed, which effectively means synchronous sharing. Conversely, if (∞, ∞, ∞) is given, then asynchronous sharing is achieved.

Thus, TACT's model requires lower specification effort than *Bayou*'s. However, the problem of mapping object updates to numerical changes of the parameters based on the system-defined notion of *conit*s (similar to records) requires additional design effort on part of the developer. From the point of view of our requirements, this is an illustration of the low abstraction flexibility provided by the system, forcing the programmer to invest additional effort to translate between application-defined abstractions and the system-defined ones.

2.3.6 Sync

Sync [18] is a repository-based sharing infrastructure that addresses the abstraction flexibility concerns raised by database systems by providing (asynchronous) sharing of *Java* objects. Its basic design idea is to provide *replicated* (shared) versions of several basic object types (e.g., `ReplicatedString`, `ReplicatedVector`, `ReplicatedDictionary`) and allow the application to automatically share object structures built with these primitives. As the following *Sync* implementation of the shared outline illustrates, programmers can define their own replicated classes by extending the `ReplicatedRecord` class, which indicates to *Sync* that the class contains replicated fields. The replicated fields must be declared public so that the system can access and automatically share them through *Java*'s (computational) reflection mechanism [13].

```
public class ReplicatedOutline extends ReplicatedRecord {
  public ReplicatedString title;
  public ReplicatedSequence sections;
}
```

In essence, the programming effort of writing a collaborative application with *Sync* consists of casting the shared application structures into the set of shared primitives supplied by the system. The underlying sharing mechanism is based on a table-driven merge algorithm, which integrates changes made by different users with a master copy maintained by a *Sync* server. Each replicated type has a default merge table, which may be overridden by the programmer, or the end user. The merge procedure runs at the server and compares the set of updates sent by a client with the set of operations applied to the master copy from the last synchronization session. As an example, consider the default table for the `ReplicatedDictionary` as given in [18]:

	Server operation (o_s)			
Remote operation (o_c)	Put(*key*)	Remove(*key*)	Modify(*key*)	null
Put(*key*)	S:∅, C:o_s	S:o_c, C:∅	S:∅, C:o_s	S:o_c, C:∅
Remove(*key*)	S:∅, C:o_s	S:∅, C:∅	S:∅, C:Put(*key*)	S:o_c, C:∅
Modify(*key*)	S:∅, C:o_s	S:∅, C:o_s	merge(o_s, o_c)	S:o_c, C:∅
null	S:∅, C:o_s	S:∅, C:o_s	S:∅, C:o_s	

Figure 2.6 *Sync*: Default merge matrix for `ReplicatedDictionary`

The rows and columns represent the operations—*put*, *remove*, and *modify*—performed by the client (o_c) and the server (o_s), respectively (*null* refers to no operation). The table provides instructions on how to handle different combinations of operations submitted by the client and the server using the same *key* parameter (operations using different parameters are deemed not to be in conflict). The notation "S:x, C:y" means that the merge algorithm should apply operation "x" to the server copy and instruct the client to perform "y" on its copy. Thus, according to the above table, if both the client and the server inserted a new element with the same key (a *put* operation) then the server operation wins: server keeps its copy (∅ means no action) and the client must execute the server *put*. If two concurrent modifications to the same entry are submitted then the merge procedure is applied recursively in the hope that the two can be reconciled at a finer granularity.

The original *Sync* implementation supports only asynchronous sharing although there are no conceptual problems to applying this approach for synchronous sharing, as a subsequent

extension has demonstrated [21]. Nevertheless, *Sync* lacks a mechanism that would allow it to provide the variety of sharing modes supported by *Suite*, for example. With respect to abstraction flexibility, however, *Sync* presents a better solution than *Suite*, because in addition to concrete data types, it can share programmer-defined types through inheritance. However, this comes at the price of exposing the internal structure of the shared types, and is, therefore, in conflict with the data encapsulation principle of object-oriented programming.

Another flexibility problem is the need to cast application structures into a fixed set of primitives with specific implementations. For example, our outline has a sequence of sections, however the *ReplicatedSequence* implementation may not be appropriate because we might need to, say, automatically renumber all subsections whenever a new section is added.

Sync does not specifically address extensibility and code reuse. Since the implementation relies on class inheritance to implement shared abstractions, adding sharing capabilities to a single-user application is equivalent to reengineering (part of) the class hierarchy. Thus, the effort involved is proportional to the number of shared classes. Furthermore, the modifications to the code are likely to be non-trivial because *Java* does not support multiple inheritance.

Since *Sync* supports the widest range of shared abstractions, it is also one of the benchmarks in our evaluation. In other words, we will show that our infrastructure can at least support *Sync*'c the range of shared abstraction.

2.3.7 Summary

Table 2.1 provides a concise summary of our survey on existing sharing infrastructures by mapping their features to our generic infrastructure requirements. A detailed explanation and justification of each entry is provided in Chapter 6, where we perform an explicit comparative evaluation of all infrastructures, including our own.

	Automation	Shared Abstraction	Sharing Modes	Code Reuse	Sharing Specification
Shared UI Systems	High	Widget, window, frame buffer	WYSIWIS only	High	None
GroupKit Shared Environmets	High	Environment	Model: synchronous/ View: flexible	High	None/Procedural
GroupKit Multicast RPC	High/Low	Arbitrary object	Synchronous/ Asynchronous	Moderate	Procedural
Colab	High	Arbitrary object	Synchronous	High	Declarative
Suite	High	Concrete types, sequence, record	Flexible synchronous/ asynchronous	Low	Parameter-based
JViews	Moderate	System-defined abstract types	Synchronous, asynchronous, user-mediated	High	Graphical
DISCIPLE	High	JavaBeans (abstract records)	Synchronous	High	Graphical
AMF-C	Low	AMF Object	Flexibile synchronous	Low	Graphical
DFS	High	Directory, File	Fixed asynchronous	High	None
Coda	High/Low	Directory, File	Fixed/flexible asynchronous	High	None/procedural
Lotus Notes	High	Database	Flexible asynchronous	Low	Parameter-based
Bayou	Moderate	Database	Flexible asynchronous	Moderate	Procedural
TACT	Moderate	Database	Spectrum	Moderate	Parameter-based
Sync	High	System-defined abstract types	Flexibile asynchronous	Low	Parameter-based

Table 2.1 Surveyed Infrastructures vs. Requirements

The basic conclusion we draw from our discussion is that, while for each of the requirements there is at least one infrastructure that satisfies it to a high degree, there is no single infrastructure that covers all of them. Moreover, very few of the systems cover a significant fraction of the requirements.

In the following chapter we present a new application model that allowed us to build an infrastructure that better satisfies the set of presented infrastructure requirements. The model combines some of the successful design approaches of previous systems with a set of new mechanisms to achieve this goal.

3. CONCEPTUAL MODEL

To establish our thesis, we first motivate and present a set of new mechanisms targeted at some of the specific limitations we have identified in current collaboration infrastructures. The main rationale behind our approach is to relax or eliminate altogether some of the assumptions traditionally built into the sharing infrastructure by replacing them with developer provided specifications. As our discussion will show, this design philosophy eventually leads to an infrastructure that is more adaptable to the needs of individual applications and significantly reduces the programming effort in developing collaborative applications. The main challenge is to keep the specification mechanisms easy to use and to ensure that they do not have a noticeable impact on application performance. In this chapter, we present the conceptual model behind our infrastructure implementation. The details of implementation are discussed in Chapter 4.

Our model has four different aspects designed to address different requirements:

- *Shared abstraction model.* The primary purpose of our shared abstraction model is to address our abstraction flexibility requirement. We introduce a pattern language that allows us to describe and share a class of objects that is a proper superset of the shared objects in existing systems.
- *Architectural model.* The primary purpose of our architectural model is to combine the higher level of abstraction flexibility of our pattern-based abstraction model with high extensibility. For that purpose, we introduce the notion of a *property handler* and describe a general approach to implementing infrastructure functions based on properties.
- *Sharing model.* The primary purpose of our sharing model is to address our sharing flexibility requirements in providing at least as much flexibility as existing systems for the set of shared abstractions defined by our abstraction model. We extend the *Suite* coupling model and introduce application layer specifications to account for the more general assumptions that our abstraction model makes about the shared objects.

- *Sharing specification model.* The primary objective of the specification model is to address our specification flexibility requirements. We present a model based on the *Suite* inheritance model and we compliment it with layer-based macro commands.

Implicitly, many of the mechanisms also contribute to the fulfillment of our automation requirement and some support code reuse. In particular, by extending the set of programmer-defined abstractions that can automatically be shared, we reduce the development effort and allow existing abstractions used in the single-user implementation to be directly reused. The architectural model enables the set of these shareable abstractions, as well as the set of sharing functions, to be incrementally extended. Also, by isolating the application-specific code related to sharing into property handlers, it promotes the reuse of both application and infrastructure code. Finally, the specification model automates the process of dynamically specifying the collaboration. For the rest of this chapter, we discuss in detail each of the above points.

3.1 Shared Abstraction Model

3.1.1 Introduction

Recall that one of the main goals stated in our thesis is to improve abstraction flexibility. The single most important factor determining the level of abstraction flexibility is the choice of a basic shared abstraction–the more general the shared abstraction the more flexibility it provides. In this respect, a system based on sharing windows, for example, would be less flexible than a system based on sharing objects because the former provides sharing only of window objects whereas the latter could potentially share arbitrary objects, including window objects.

In general, if an infrastructure provides sharing at the level of the basic programming primitives provided by the underlying programming language (e.g., variables in procedural languages, objects in object-oriented languages), it does not place any restrictions on the implementation and, therefore, provides the highest level of abstraction flexibility. Another aspect in evaluating the abstraction flexibility of an infrastructure is whether it supports the sharing of concrete or abstract data types—a system that supports the sharing of abstract data types can also support the sharing of concrete types but not vice versa.

Considering the current prevalence of the object-oriented approach in development platforms, we have focused our efforts on providing flexible object sharing facilities as the basis for supporting flexible collaboration. Thus, our ideal system should support the sharing of arbitrary objects. However, reconciling this goal with object programming principles presents some non-trivial problems.

The main issue we face is the apparent conflict between data encapsulation and object sharing–the former mandates that object state be kept private and accessed through programmer-defined methods, whereas the latter needs to know the object state in order to share it. As it turns out, if we respect data encapsulation and keep the state private, it is fundamentally impossible to implement the automatic sharing of arbitrary objects. To illustrate this point, consider the original class definition we used in our outline example:

```
public class Outline {
  public String getTitle();
  public void   setTitle(String title);

  public Section getSection(int i);
  public void    setSection(int i,Section s);
  public int     getSectionCount();
  public void    insertSection(int i,Section s);
  public void    removeSection(int i);
}
```

Figure 3.1 Example Outline implementation

Assume that we have the two *Outline* instances that have diverged (e.g., users have been off-line for a while) and we need to bring them back into a consistent state. Clearly, applying a generic mechanism, such as broadcast methods, alone would not solve the problem—the sharing mechanism would need to have at least some basic understanding of the semantics of the public methods in order to accomplish the task. However, deriving such knowledge about the object automatically would amount to solving the halting problem, which is fundamentally impossible. Consequently, in the general case, it is not possible to provide fully automatic flexible object sharing while preserving data encapsulation. Therefore, our realistic goal is to support the flexible automatic sharing of a wider range of objects than currently possible. Since knowing the logical structure of the shared entity is a precondition for implementing fine-grained sharing services, we have focused an important part of our efforts on providing a more expressive mechanism for deriving the (logical) structure of

shared objects (as opposed to the in-memory physical structure defined by its internal variables)

Existing infrastructures have taken two different approaches in dealing with the problem of obtaining the structure of shared objects. The first one, employed by systems such as *JViews* and *AMF-C,* is to get around it by defining a low-level communication protocol, and require the application developer to adapt the shared objects so that they can communicate through this system-defined protocol. The obvious advantage here is that the infrastructure can accommodate the sharing of any object as long as it adheres to the protocol. In practice, however, the additional effort is often non-trivial, and requires modifications that are spread all over the single-user code. Moreover, since the infrastructure deals with a single stream of events, it cannot, for example, make a distinction between updates affecting the shared object and events affecting its appearance. Thus, the infrastructure either does not support different levels of sharing (*JViews*), or the programmer must manually establish different communication channels and manage their coordination (*AMF-C*). Overall, this leads to less automation and code reuse than the alternative approach, based on shared programming abstractions.

Infrastructures, such as *XTV*, *Suite*, and *Sync*, opt to relieve the programmer from the potentially expensive task of code adaptation by supporting shared primitives that can be directly incorporated into the application. Although these systems share a common approach, they provide different levels of support. In *XTV*, only a single shared type (shared window) is supported, whereas *Suite* and *Sync*, in addition to several basic types (integer, string, etc), also support complex hierarchies of programmer-defined types, built through records, sequences, and tables (*Sync* only). As already discussed in the previous chapter, *DISCIPLE* supports sharing of *JavaBeans*—an altogether different set of objects. The common advantage among these systems is that they provide a high level of automation because the shared programming abstractions are directly employed by the application. Their common limitations stem from the fact the application *must* use the provided shared abstractions to achieve sharing. In case those are not suitable, the developer is left with the choice of implementing custom-built abstractions, or adapting the application to the supported abstractions. The latter process can be especially difficult if the starting point is an already implemented single-user version.

Our goal, as shown on Figure 3.2, is to have a shared abstraction model that subsumes the existing ones and also shares an extensible range of objects not currently supported by other systems. To achieve this goal, we have developed a shared abstraction model based on programming patterns, which we also refer to as *pattern-based object model*.

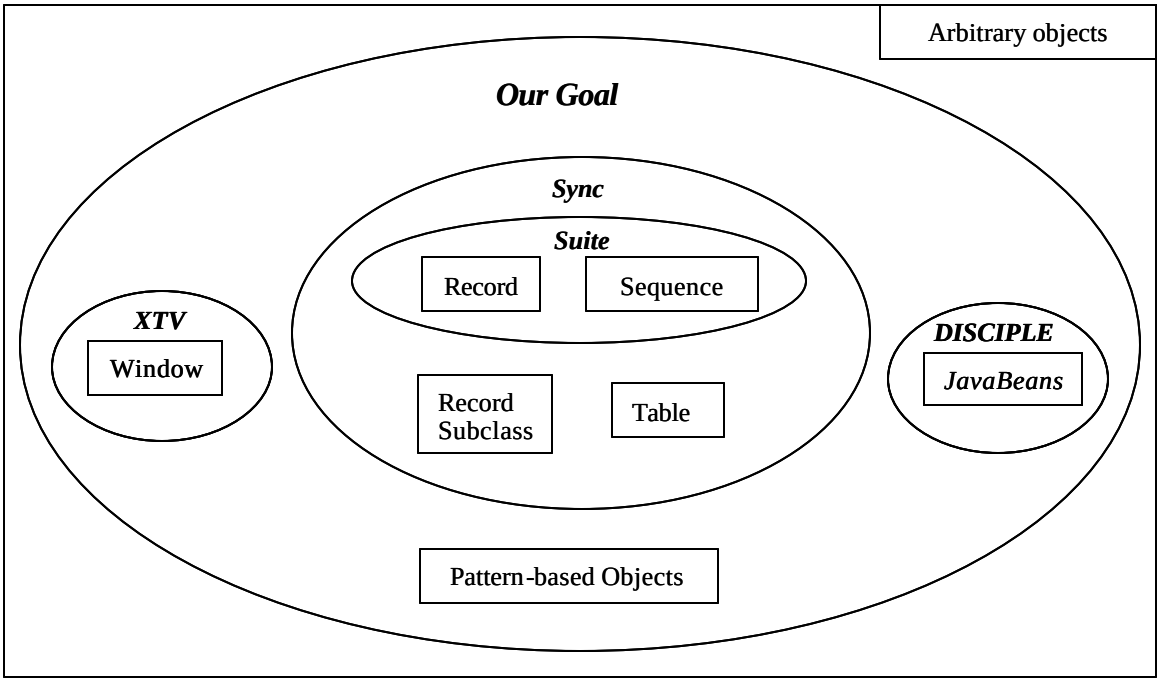


Figure 3.2 Sharing infrastructures and their shared abstractions

3.1.2 Pattern-based Object Model

One of the consequences of data encapsulation is the fact that objects must provide public methods that permit other objects to access and update its internal state. Furthermore, if different aspects of the object's state can be manipulated independently, it is highly desirable that they be accessible through separate methods. For example, in the implementation of our outline object, we provided separate methods for updating the title and the list of sections (`setTitle` and `setSection`), instead of one generic method, e.g. `set`, which handles all updates. In essence, this separation of responsibilities among the methods reflects the logical structure of the object without revealing its implementation.

To make this object structure apparent, good programming practices dictate the use of sensible method naming conventions—or *programming patterns*—that help programmers determine the likely purpose of a method without looking into the code itself. In our outline example, we named the method responsible for updating the title `setTitle` to suggest its

purpose. Furthermore, a group of methods dealing with a particular aspect of the object's state would have certain naming commonality to emphasize that fact. For that reason, we named the method that gives the current title of the outline as `getTitle`.

Certainly, other reasonable (and more complicated) programming patterns have been in use for a while. However, these programming patterns have remained largely informal and, until recently, there has been no consistent effort to standardize and take advantage of them for the purposes of automated development. *Java*, through the introduction of the *JavaBeans* platform, has made the first step towards the more general use of these patterns.

3.1.3 Programming Patterns in JavaBeans

The main goal of *JavaBeans* is to define a generic component-based architecture for Java programs. The whole approach is based on the use of the *Java Reflection API* to achieve run-time discovery, composition, and customization of application components. As a strongly typed, interpreted language, *Java* must retain a lot of the original information provided by the programmer, such as method signatures and inheritance relationships, to perform run-time checks. Furthermore, all of this information is kept in its original string form because of *Java*'s late binding mechanism, which loads class definitions on demand. Thus, the reflection API simply provides a standard means of accessing this publicly available information. Building on this relatively low-level API, *JavaBeans* defines a more abstract *introspection* mechanism that automates the handling of application objects, called (*Java*) beans, which are structured according to a set of standard conventions.

The most important requirement for a bean object is to advertise its structure in the form of properties. According to the *JavaBeans Specification*, 'properties are discrete, named attributes of a *Java* bean that can affect its appearance or its behavior'. This, however, is a rather broad and vague definition, which does not explicitly address the issue of *how* a property is defined. Therefore, by summarizing the discussion in the *JavaBeans* specification, we use the following, more formal, definition:

*An **object property** is a named attribute of an object that adheres to a predefined semantics and can be manipulated through a set of dedicated methods.*

One straightforward way of specifying an object property is to define an interface containing the property methods and to require objects that have such a property to

implement the interface. For example, we could use the following interface to define a title property and allow our outline object to advertise the fact that it has such a property by implementing the `TitleProperty` interface as follows:

```
public interface TitleProperty {
  public String getTitle();
  public void setTitle(String title);
}
public class Outline implements TitleProperty {
  // class implementation remains unchanged
  public String getTitle();
  public void setTitle(String title);
  …
}
```

Similarly, we could define interfaces to *explicitly* advertise other properties that the outline might have: date, author, etc. However, to describe the properties of a complex object, such as a user interface window, we would need a large number of "small" interfaces—interfaces containing only a few methods—that would appear very similar. This would clutter the code and may well make the task of understanding it more difficult. More importantly, programmer-defined interfaces would not be known to the infrastructure. At the same time, an interface only provides a syntactic contract between two objects and does not guarantee anything about their behavior.

Building on this fact, *JavaBeans* turns to programming patterns to define object properties *implicitly* rather than explicitly through interfaces. *JavaBeans* recognizes two types of properties–*simple* and *indexed*. A simple property is defined by a couple of 'get' and 'set' methods of the following form:

```
<PropertyType> get<PropertyName>()
void set<PropertyName>(<PropertyType> value)
```

The 'get' method corresponds to a read operation and returns the current value of the property, whereas the 'set' method defines a write operation that assigns a new value to the property. Thus, if a matching 'get'/'set' pair is found during the object analysis, then a read-write property named *PropertyName* of type *PropertyType* is discovered. In some cases, one of the methods may be absent in which case the property is considered as write-only/read-only respectively. Returning to our example, the pair of `getTitle/setTitle` methods defines a simple property called *title.*

Indexed properties are a straightforward extension of simple properties and approximate standard array semantics in procedural languages. Whenever a simple property of an array

type is discovered, it is considered to be an indexed property. The object is then searched for a second pair of get/set methods that manipulate individual elements by their index. In the following class definition, the four shown methods define an indexed property called section of type *Section*.

```
public class Outline {
  public Section[] getSection();
  public void setSection(Section[] section);
  public Section getSection(int i);
  public void setSection(int i, Section section);
  …
}
```

In essence, the described *JavaBeans* programming patterns for simple and indexed properties can be interpreted as implicitly defining two families of interfaces, whose names and member methods are derived using the get/set naming convention. Thus, the programmer gets the benefits of specifying object properties without the overhead of managing a multitude of interfaces and the restriction of using only infrastructure-defined interfaces.

However, despite this added expressive power, *JavaBeans'* property specification mechanism has its own expressive limitations. To understand the problems, let us first consider the impact of hardwiring the syntax of the get/set pattern into *Java*'s introspection mechanism. The most important consequence is that legacy code must be rewritten to comply with the exact specification. This is a non-trivial issue, given that even current versions of the *Java* APIs are not fully compliant. At the same time, well-written systems tend to follow similar naming conventions but may use other keywords. For example, instead of 'get', developers may have used other verbs to describe the reading of an attribute–'read', 'check', etc. Similarly, write operations may be represented by methods starting with 'write', 'update', 'reset', etc., instead of 'set'.

While it is possible to accommodate the above cases by providing additional information to the introspection mechanism through 'BeanInfo' specification classes, in practical terms, this means that the programmer must write an additional *Java* class for each class containing exceptions to the standard get/set syntactic rules, and explicitly point the system to the correct methods. The regular and the specification classes are linked by a naming convention, which allows the introspection mechanism to automatically discover the specification. For example, the BeanInfo class for the Outline class, would be named OutlineBeanInfo and would implement the system-defined BeanInfo interface. During the introspection process,

the system always checks for the presence of a specification class, and, if one is found, its content supersedes any information derived implicitly by the system.

However, there are at least two problems with this approach, which severely limit its applicability. First, using a procedural mechanism to write specifications is rather clumsy and typically results in the writing of many lines of routine code that are not directly relevant to the application's functionality. Second, the specification effort is proportional to the number of classes affected. In the worst case, the number of specification classes could be proportional to the total number of classes comprising the application.

Another, more important, expressive limitation of *JavaBeans*' properties is the fact that their *semantics* is hardwired into the introspection mechanism. Given our goal of automated sharing of arbitrary structured objects, the existing *Java* support for properties is not general enough for our purposes. Intuitively, it should be possible, for example, to describe our section sequence as some kind of *JavaBeans* property. However, there is no suitable way to describe the dynamic nature of this property–its ability to incrementally change its structure through the addition/removal of individual elements.

In summary, the current *JavaBeans* property model lacks the expressive power to describe several useful kinds of structured objects and, therefore, a collaborative infrastructure based on this model would lack the abstraction flexibility to share such objects. Therefore, we present an extensible framework for describing and using object properties that remedies the discussed limitations.

3.1.4 Generalized Properties and Programmer-defined Patterns

In an effort to gain expressive power, we separate the problem of specifying a property from its actual use. For that purpose, we introduce a declarative specification language, which allows programmers to define object properties in terms of corresponding programming patterns. We refer to the process of identifying the properties of an object as (*property*) *introspection*, and to the infrastructure service object that implements it as `Introspector`. Our actual specification language, described in detail in the next chapter, is based on *XML*[3]. However, for the purposes of this discussion, we use a simplified syntax that is more concise and easier to read.

[3] http://www.w3.org/XML/

We adopt the de facto standard mixedCase naming convention as the basis for our pattern analysis. Specifically, we assume that method names consist of one or more tokens. The beginning of each token is marked by a capital letter following a small letter or, in case there is more than one consecutive capital letters, by a capital letter followed by a small letter. For example, `getHTMLGenerator` is decomposed into three tokens (`get-html-generator`). The rest of this discussion considers pattern matches at the token granularity.

The first step in the property specification is to define method (signature) patterns, which select the candidate methods to be matched against the constraints of the property definition. The method patterns are based on a canonical string representation of method signatures of the following form

```
pattern_method = <return_type> method_name(arg1_type,…,argN_type).
```

Method patterns contain free pattern variables that are assigned string values and, in our notation, are separated by a pair of '<', '>' symbols. For example, to define the standard *JavaBeans* 'get'/'set' methods we use the following declarations:

```
getter = <GetType> get<PropName>()
setter = void set<PropName>(<SetType>)
```

Pattern variables are assigned upon the completion of a successful match, and contain the maximum length match, which may span several tokens. For example, if the above getter declaration is matched against the method `int getXPos()`, the values of the pattern variables would be as follows `GetType == "int"` and `PropName == "XPos"`.

The second step is to define the conditions, or *constraints*, under which candidate methods are grouped together to define properties. A constraint is a simple test of equality between the string values of two assigned pattern variables:

```
pattern_method1.pattern_variable1 == pattern_method2.pattern_variable2
```

Candidate methods that have satisfied all the constraints of a property definition are called *pattern methods*. In addition to these, each property also has a naming rule based on which the introspector can assign a name that distinguishes among all the object properties satisfying the same definition:

```
name = pattern_method.pattern_variable | literal
```

The `literal` option refers to the ability to specify a fixed name for the property. This is necessary to deal with cases in which the pattern variables do not provide suitable

information to generate a unique property name. Evidently, this also implies that such a property definition should not match more then one property per object.

To illustrate the whole specification of a property, consider the following complete declaration of the standard *JavaBeans* properties:

```
property
  type = simple
  methods
   getter = <GetType> get<PropName>()
   setter = void set<PropName>(<SetType>)
  constraints
   getter.PropName == setter.PropName
   getter.GetType == setter.SetType
  name = getter.PropName
end
```

The specification states that a property of type “simple” is defined whenever two pattern methods can be found such that they match the *getter/setter* patterns, and their respective pattern variables satisfy the given constraints. The last line specifies the naming rule for the property. Note that, at this point, a property is a purely syntactic construct that can be discovered but not yet interpreted by the infrastructure. Thus, the string specified by the `type` clause is merely an identifier that has no bearing to the interpretation of the property.

Recall that one of the shortcomings of the *JavaBeans* model was the inability to provide alternative patterns for the same type of property. To show how we handle this issue, consider describing the special case of properties of `boolean` type. As an exception to the standard *JavaBeans* naming conventions, it is permissible for the getter method to have the following form:

```
boolean is<PropertyName>()
```

Describing this exception in our model is straightforward–all we need is an alternative definition for the getter method and the rest of the definitions will work as before:

```
getter = <boolean> is.PropName.()
```

Other, programmer-defined patterns can be described in a similar fashion. For example, applying the following definition to our example *Outline* class produces a sequence property called *Section*.

```
property
  type = sequence
  methods
   insert = void insert<PropName> (int, <InsType>)
   remove = void remove<PropName> (int)
   lookup = <GetType> get<PropName>(int)
```

```
    set    = void set<PropName> (int, <SetType>)
    count  = int get<PropName>Count()
  constraints
    insert.PropName == remove.PropName
    insert.PropName == lookup.PropName
    insert.PropName == set.PropName
    insert.PropName == count.PropName
    insert.InsType == lookup.SetType
    insert.InsType == lookup.GetType
  name = insert.PropName
end
```

By default, our prototype implementation comes with three basic property definitions for *simple*, *sequence*, and *table* properties that allow us to simulate the abstractions provided by our benchmark systems. Furthermore, versions of the *sequence* and *table* definitions are tailored so that they describe properties in the standard (and widely used) `Vector` and `Hashtable` classes, respectively. We discuss the role of property specification versions in the next chapter in infrastructure implementation.

In summary, the described pattern description language allows, unlike any of the mechanisms in current infrastructures, the description of the structure of our original *Outline* object implementation with no modifications. Also, we showed that *JavaBeans* properties are a special case of the use of patterns that can be succinctly described in one property definition. Next, we consider the relationship between patterns and interfaces.

3.1.5 Patterns vs. Interfaces

As it turns out, interfaces are another special case of the use of patterns—one in which method patterns contain no free variables (consequently, the constraint clause is also empty). As an illustration, consider the description of *Sync*'s *ReplicatedSequence* class:

```
property
  type = sequence
  methods
    insert = void insertElementAt(int, Object)
    remove = void removeElementAt(int)
    lookup = Object getElement(int)
    set    = void setElement(int, Object)
    count  = int size()
  constraints
  name == 'Element'
end
```

The above pattern-based specification is logically equivalent to the following *Java* interface-based specification in the sense that whenever an object implements the `Sequence` interface below (or extends an object that does) it will also have the above *sequence* property.

```
public interface Sequence {
  void insertElementAt(int, Object);
  void removeElementAt(int);
  Object getElement(int);
  void setElement(int, Object);
  int size();
}
```

To better understand the advantage of using patterns over interfaces, consider the implementation of a shared outline and a shared sequence using patterns and interfaces. For the purposes of this example, assume that the task at hand is to replay a remote insert operation and we want to compare the cost and flexibility of the two approaches. Using patterns, the insert can be replayed in two lines of code:

```
void replayInsert(Object target, int pos, Object arg) {
  Method insertMethod = Introspector.getMethod(target,“Element”,“insert”)
  insertMethod.invoke(target, {pos, arg})
}
```

The `Introspector` refers to the system object, which performs the identification of properties and provides an API to access the results from the pattern analysis. Thus, the `getMethod` call asks for a reference to the method matching the “insert” method pattern, of property “Element” of object `target`. The second line is a reflective invocation of the insert method on the target object with the given arguments. We should note that even if the set of patterns describing sequences extends further, only additional property specifications would be needed—the above code would still work.

In contrast, the interface-based solution requires two interfaces—one including all the “section” methods from the `Outline` class and another including the `Sequence` ones shown above. An interface-based solution would need to separately handle each of the two interfaces, which would require more coding. More important, however, is the fact that references to the interfaces are hardwired into the code and would need to be modified every time a new “similar” interface is introduced due to variations of the programming patterns.

Thus, in terms of expressive power, our specification language completely subsumes both interfaces and *JavaBeans* properties and automatically supports an extensible set of objects not covered by existing mechanisms. Specifically, the logical structure of any object that uses a combination of the set of patterns given to the system would automatically be recognized. Furthermore, by using a specification mechanism outside the programming language we improved code reuse and automation by allowing new object types to be accommodated with

little or no extra effort on part of the developer, and in particular without writing any extra code.

3.2 Architectural Model

3.2.1 Property Handlers

So far, our object model has addressed the issue of discovering the structure of the objects to be shared. However, we also need a mechanism that puts together the object and the sharing mechanism such that we satisfy our extensibility requirements. Recall that systems based on a common communication protocol exhibit high extensibility at the expense of automation, whereas systems based of shared programming abstractions tend to offer the reverse combination of features—high automation and low extensibility. Consequently, a primary objective of our architectural model is to combine the higher level of abstraction flexibility of our pattern-based model with high extensibility without increasing the development effort.

To motivate our approach, let us continue with the analysis of our *Outline* object. So far, based on two property specifications we can discover that the outline consists of a *title* (a string value) and a *sequence* (of `Sections`). However, to discover the complete hierarchical structure of the shared outline, we need to recursively analyze each of the *Section* objects until we reach leaf nodes of atomic types. For that purpose, the infrastructure needs to know how to obtain the current value of each type of property. However, hardwiring such information into the infrastructure implementation would largely defeat the benefits of our pattern-based object model—the infrastructure would be able to discover new types of properties based on new specifications but it would be unable to interpret them, e.g., get their current values.

To resolve this problem, we introduce the notion of a *property handler*—a method (procedure) that implements a system-defined operation, such as reading the current value, for a particular property type. Since handlers are defined on a per-property basis, the natural place for their specification is the property definitions, where we add a `handlers` clause as in the following example:

```
property
  type = simple
  //same definition as before
```

```
  …
  handlers
    read = colab.SimplePropertyReader
    // more handlers may be defined
end
```

The interpretation of the handler declarations is as follows: the left-hand side gives the name an abstract operation on the property, whereas the right-hand side points to a *Java* class that implements a specific method known to the infrastructure. In this example, the *read* handler implements the method

```
public Object[] getValue(Object target, Property property).
```

The two arguments in the `getValue` methods permit the writing of generic, class-independent property handlers. The *target* argument identifies the run-time object on which the operation is to be performed, while the *property* argument makes available to the programmer the results of the pattern matching performed by the introspection process. In particular, it provides references to all the method matches discovered. As an example, consider the following generic implementation of *read* handler for simple properties:

```
public Object[] getValue(Object target, Property property) {
     Method getMethod = property.getMethod("getter");
     Object[] value = new Object[1];
     value[0] = getMethod.invoke(target, null);
     return value;
}
```

The `property.getMethod("getter")` call returns a reference to the a method matching the *getter* pattern (`getter = <GetType> get<PropName>()`) from the specification of a simple property (and satisfies all applicable constraints). The two arguments of the `invoke` call represent the instance on which the method should be invoked and the list of arguments (in this case none), respectively. Once the *read* operation is defined on all properties, we can apply the property analysis in a recursive manner to discover the entire hierarchical structure as shown below:

```
ObjectWalk(object)
  if object == null || object in visited list
    return
  <add object to visited list>
  <process object>
  for each property p_i of object do
    reader = lookup read handler for p_i
    if reader != null
       values[] = reader.getValue(object,pi)
       for each v in values
         ObjectWalk(v)
       end
```

```
      fi
    end
  end
```

In essence, the generic `ObjectWalk` procedure shown above uses the introspection analysis to decompose objects into properties and to delegate the problem of interpreting a property to a property-specific handler. Thus, property handlers provide a level of abstraction that isolates the infrastructure from the actual set of properties used in the application. Since the binding of an infrastructure service and the handlers is performed at run time through the property specifications, a handler can be reused for multiple objects. For example, the *sequence* read handler would work with both the `Outline` and the `Section` objects (the latter needs to maintain a sequence of subsections). The binding of the application objects and handlers is implicit via programming patterns.

3.2.2 Property Handlers vs. Requirements

To conclude our discussion on property handlers, let us examine how property handlers map to our stated requirements:

- *Extensibility:* The use of handlers completely separates the shared object from the sharing functions by isolating collaboration-aware code into a separate object. Combined with the declarative binding mechanism, which permits run-time composition, we can reasonably claim that handlers support extensibility.
- *Automation:* Depending on the consistent use of programming patterns, property handlers are likely to significantly reduce the programming effort involved in sharing new abstractions. This is based on the fact that a property handler can be directly reused in sharing objects that are unrelated in the class hierarchy. In other words, application objects do not have to implement infrastructure-defined interfaces in order to be shareable. Consequently, the overall effort to implement sharing of a set of abstractions is likely to be smaller compared to an implementation based solely on interfaces. In the worst case, when no programming pattern is reused (which also implies that all objects are unrelated) the infrastructure must, in essence, provide a custom implementation by dealing with each individual abstraction separately. In this situation, a property-based implementation provide requires approximately the same effort as a standard object-based implementation that uses classes and interfaces.

- *Code reuse:* handlers support both compiled code reuse and incremental collaboration awareness. There are three different aspects of our support for reuse: application code reuse, infrastructure code reuse, and handler code reuse.

  Using handlers allows single-user application code to be completely reused because the functions related to sharing are implemented separately in the handler code. Furthermore, if new properties and corresponding handlers are needed, their incremental addition will be transparent to the infrastructure (as well the application) and will allow the reuse of infrastructure code. Finally, the handler code itself is also readily reusable—most patterns are a function of programming style and are independent of the application semantics. In other words, most patterns, such as the *sequence* pattern defined in the previous section, are so generic that are likely to be used in many applications. Hence, the property handlers written for such patterns (such as a *read* handler for *sequence* properties) can be reused in any application that uses the pattern at virtually no cost.

3.3 Sharing Model

The basic units of sharing in our model are object properties. We assume that the sharing of each individual property of the shared object structure can be controlled independently of the rest of the shared structure. Here, we present a sharing model that addresses our sharing flexibility requirements at the finest granularity—properties. In the following section, we complement this basic sharing model with a sharing specification model that satisfies our sharing specification requirements. We should note that the model we describe here is not sufficient to handle infrastructure services such as access control or (pessimistic) concurrency control. Such control mechanisms require that the infrastructure receive announcements of intended operations before they are performed. While we do not address the issue in this work, our event model design is compatible with future planned extension and we describe our ideas in this respect in Chapter 7.

Recall that our benchmark with respect to sharing and specification flexibility is *Suite.* Therefore, our sharing model is based on a generalization of the *Suite* coupling model that was described in the previous chapter. One fundamental difference between our approach and *Suite*'s is that we do not assume full control over the editing of the shared object.

Therefore, we need an explicit event model that provides a generic communication protocol between the application and the infrastructure. Given that properties are our basic unit of sharing, it is natural to consider events that encode property updates as the basis of our communication model.

3.3.1 Property Events

This approach is similar to the event model adopted by *JavaBeans* where all property changes are communicated through instances of the system-defined *PropertyChangedEvent* class. It encodes the source object of the update, the name of the property affected, as well as the old and new values for the property. Since in *JavaBeans* there is only one operation modifying the property values–*write*–this single class is sufficient to meet all communication needs.

In our case, we need a more general approach and, therefore, our event model assumes that *property events* encode corresponding operations performed on a particular property, i.e., an invocation on one the property's methods. Thus, an update event is a four-tuple, which contains:

- A global identifier of the object on which the operation is performed.
- The name of the property affected.
- The name of the pattern method invoked as defined in the property specification (e.g., "insert", not a reference to the *insertElementAt* method)
- A list of arguments (perhaps empty).

For example, inserting a new section into the outline would look as follows:

```
<"Outline", "section", "insert", {2, section}>.
```

Given this information, and our property-based introspection analysis of the object, it is relatively straightforward to replicate the encoded operation at a remote site (the details of this process are discussed in the next chapter). While this mechanism may resemble the static broadcast methods of *Colab*, it is much more flexible because, depending on the collaboration policies in place, the sharing infrastructure may dynamically choose to delay, transform, merge, or discard property operations to achieve the desired sharing mode.

As an added advantage, this mechanism facilitates the adaptation of application-specific event models for the purposes of object sharing. Usually, event processing in single-user

applications is asymmetric—an object issuing notifications of a particular type is not designed to receive and process such notifications from peer objects. For example, a standard `TextField` object issues `TextEvent` notifications but it is not designed to react to such events. Thus, there are two inherently application-specific operations that the infrastructure must perform to adapt a single user event model. First, upon acquiring an event, it must transform it so that it can be transmitted and replicated remotely, and second, it must transform the event into a suitable method invocation at the remote site. By choosing the above-described format for property events, we effective merge the two transformations into a single one that is performed at the original site by an event adapter. Once transmitted, the property event can be applied at the remote replica without knowledge of the adapted event model. Hence, the adaptation of an application event model is reduced to writing a single adapter that maps application events to property events.

3.3.2 Sharing Parameters

In this section, we discuss the rationale and meaning of the sharing parameters, while the implementation details of how these are put together in practice are discussed in Chapter 4. We should note that the model we describe here is designed to address the issue of flexible object sharing by managing the updates notified by multiple object replicas. However, it does not address issues related to controlling access to the object, such as (pessimistic) concurrency control and access control (possibly preventing updates in the first place). These are the subject of our future work and do require an extension of this model that allows the infrastructure to control updates before they have occurred.

We distinguish among three phases in the handling of property events: *acquisition*, *processing*, and *installation* (Figure 3.3), and each of these phases is controlled by its own parameters. In the acquisition phase, a description of the user update is obtained by the infrastructure. In the processing phase, the infrastructure buffers, transforms, and communicates the update to the remote parties based on the current sharing policy. Finally, in the installation phase, the update is merged with the current state of the remote object to which it is delivered.

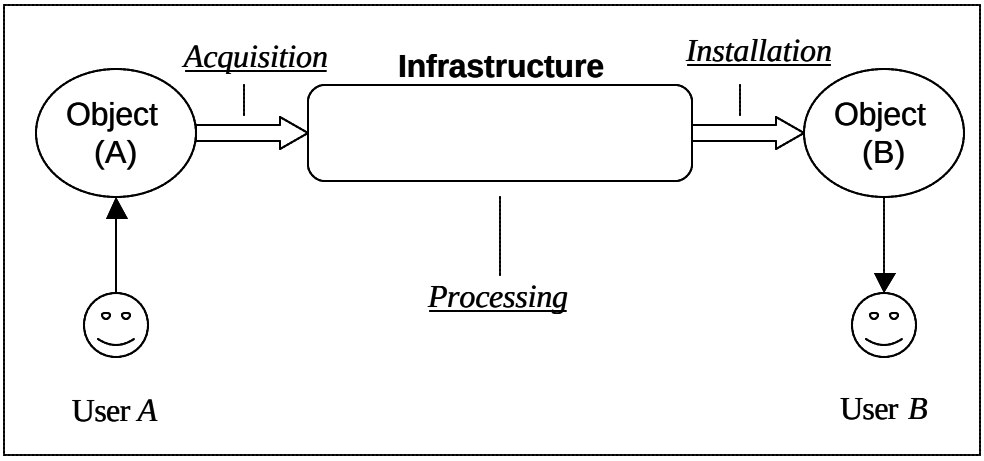


Figure 3.3 *Phases* of update handling

We associate with each shared entity four different parameters: *Transmission, Correctness, Acquisition,* and *Installation*. The first two we inherit from the *Suite* model and control the processing phase. The last two parameters form our extension to the model and control the acquisition and installation phases of the event handling, respectively. We refer to the combination of sharing parameters regarding a shared entity as a *collaboration* (or *sharing*) *policy*.

Like *Suite*, the processing parameters have two instances—one placing restrictions on *outgoing* events and one on *incoming* events specified by the sending and receiving user, respectively. Like *Suite*, we expect these to be different so we also have a *Suite-like* reconciliation mechanism based on conservative matching. By conservative we mean that of the two versions of each processing parameter (outgoing and incoming) we pick the one that has a higher value and, thus, come up with an *effective* collaboration policy. Realizing that an outgoing policy that is more liberal (sends out more events) than its incoming counterpart would lead to the communication of events that will be held at the receiver site for delivery, we perform the matching at the sending site to avoid sending such events in the first place. Hence, the receiving site does not perform filtering of incoming events but proceeds directly to install them. Performing the policy matching at the sender, as *Suite* does, implies that policy changes must be sent to the corresponding user(s) every time a user modifies the incoming policy. This, however, is a good trade off because policy updates are much less frequent than object updates.

Let us now describe the values and the meaning of each parameter in our model.

- *Transmission:* This parameter has the same purpose and semantics as *Suite*'s transmission parameter. It controls the transmission of updates based on the communication operation performed on the shared entity and has four possible values: *Increment, Complete, Scheduled,* and *Transmit*. As in *Suite,* each individual update is an *Increment* operation. A *Complete* operation is executed whenever the user has indicated that he is finished editing the value, e.g., by hitting `<tab>` or `<return>`. (Note that, unlike *Suite,* we have no control over the user interface. Therefore, for application-specific operations such as *Complete* we rely on the programmer to issue a *synchronization event*, which is discussed in the next section.) The concrete mechanism by which this is accomplished is described in Chapter 4. A *Scheduled* operation is triggered by timer expiration and has two parameters—execution time and a period. The semantics here is that the operation is first triggered at the specified (wall clock) time and it is then triggered periodically. Thus, this is a combination of *Suite*'s *TPeriod* and *TTime* time-based operations. The *Transmit* operation is executed whenever the user explicitly requests it by pressing a `<transmit>` button provided by the infrastructure.
- *Correctness:* This parameter also has the same purpose and semantics as in *Suite.* Its possible values (in increasing order) are *Raw, Parsed, Validated,* and *Committed*. By default, any updated value is *Raw*, unless it has undergone a successful syntactic check after which it is elevated to *Parsed*. If the value has also passed a check for semantic correctness, it becomes *Validated*. As with *Complete* operations, we rely on the application to issue appropriate synchronization events to tag updates as *Parsed* or *Validated*. As before, *Committed* values are explicitly designated by the user by executing a *commit* command, i.e., selecting an infrastructure-provided `<commit>` menu item.
- *Acquisition:* This parameter controls the method used to obtain the replica update whenever the infrastructure is notified that one is (potentially) available. By update we mean any change in the observable state of the replica. The update notification usually comes from user actions but it can also be triggered by a timer mechanism. Currently, we distinguish among four different acquisition methods: *Read, Log, Effective Log,* and *Diff*. To illustrate the differences among them let us consider the simple scenario of a user

editing the title of our outline through a text field and see how the different combinations of *Transmit* and *Acquisition* parameters lead to different results.

Assume that a user has just typed a character and the infrastructure got notified, say, by automatically capturing an event from the window. How should the infrastructure record the change? In *Java*, for example, the `TextEvent` object does not provide the details of the operation but merely informs that one has occurred. Therefore, one option is to record the end result of the user action by obtaining the current value of the field. In other words, whenever it receives a `TextEvent` notification, the infrastructure uses a *read* handler to obtain the current value of the text field and subsequently installs it remotely using a corresponding *write* handler. We specify this option through by setting the acquisition parameter to *Read*.

While this approach is attractively simple, it does not work very well when the field is concurrently edited by multiple users and we would like to merge their work. For instance, one user inserts a character in the beginning while another adds another one at the end of the title. In all likelihood, users would like to see their changes merged rather than giving preference of one over the other. More generally, knowing the exact fine-grained operations performed greatly increases the chances of the infrastructure being able to automatically detect (syntactically) non-conflicting updates and merge them according to user expectations. Therefore, the programmer might develop a text field object that automatically notifies the exact insert/delete operation performed by the user. (Alternatively, the insert/delete operation could be derived by a listening object that keeps track of previous state.)

By all means, once an incremental update is available, the logical choice for the acquisition parameter is *Log*-ing. It tells the system to record and later on (depending on the current transmission/correctness parameters) replay the event remotely. Thus, the main conceptual difference between using *Read* and *Log* options is that the former specifies the transfer of the entire state of an entity (property, object, application), whereas the latter installs only incremental changes. The latecomer accommodation problem discussed in Section 3.5.2 below provides an example of the differences at the application level.

Over time, however, the log of operations can become rather long if, say, one of the users is off-line for a prolonged period of time. Therefore, log-based approaches, such as *Chung*'s [4], provide mechanisms for compressing the log by removing operations whose effects will be undone by subsequent operations. We refer to this method as *Effective Log*. In the following chapter we describe a generic scheme that performs *effective logging* for property operations. Although it is not as comprehensive as *Chung*'s scheme, it requires only minimal specification effort on part of the developer. Our model also allows for an external logging module to be plugged in. For that purpose, the developer must specify a `log` property handler for each of the shared object properties. If the current acquisition method is *Log* and the infrastructure finds that a `log` handler is specified, it simply delivers the event to the handler. If the goal is to employ *Chung*'s logging service, the property specification should point to an object that can translate property events into the generic model used by *Chung* and deliver them to remote users as appropriate. In addition to remote delivery, the external service logging must also take responsibility of translating back into property events so that updates can be applied correctly.

The log-based approach assumes that it can obtain updates at a fine granularity from the application. However, in many cases the application does not provide a suitable notification mechanism to tap into. In that situation, using *Diff*-ing may provide a good solution. *Diff*-ing refers to the process of automatically deriving the fine-grained operations from two snapshots of the object's state. While the idea of diff-ing is not new, implementing object-based diff-ing presents a challenge and we are unaware of any work that presents a generic solution. However, our pattern-based approach has enabled us to come up with a new solution that is generic enough to make it a plausible choice. We defer the detailed description of this solution to Section 3.3.5.

- *Installation:* Once an update is acquired and processed, it needs to be installed on the remote object. We distinguish among three different ways in which this can be achieved: *Replay, Real-time Replay,* and *Merge*. The *Replay* option is the simplest choice—the operation is replayed on the remote object. However, depending on the intent of the collaborators, this may not be the best option. Suppose we have a late-

coming participant who wants to replay the sequence of changes made prior to his joining of the collaboration. If the operations are simply replayed one after the other, the user will likely have a "fast forward" experience in which minutes of collaboration are compressed into seconds. Due to network jitter, similar problems can arise on a smaller scale [Kum, Gutwin] and disrupt real-time collaboration, e.g., a slow, smooth mouse gesture can be perceived as random jumping of the mouse pointer because numerous small movements are delivered in a short burst and, consequently, replayed so fast that they are imperceptible by the user. Also, to maintain interactivity, end-to-end delay must also be managed to avoid, for example, a situation in which a mouse gesture noticeably lags the voice narration and creates confusion. We refer to this time-controlled replay process as *Real-time Replay*.

The *Merge* installation option refers to the use of a specialized merge procedure that integrates the update with the current version of the object. In other words, this is necessary when the simple replay of the updates (a trivial form of merging) is no longer sufficient. This type of installation is commonly used in asynchronous sharing when two, or more, replicas have diverged and conflicts only become apparent during the installation phase.

3.3.3 Synchronization Events

Our sharing model also defines *synchronization events*, which are meta-events that carry sharing information about the property events in the system and support the sharing parameters described above. To motivate the need for such events let us consider the following sharing scenario: users *A* and *B* are editing a together a form, which consists of a number of text fields. They want the system to log their changes and communicate them only when a change is *complete* that is, the user has moved from one text field to the next. To implement this, the infrastructure must buffer every character insertion and deletion in a queue because these are *incremental* changes. However, it also needs to know when all of these changes become a complete change and flush the queue. The end-of-editing event is inherently application-specific: in our example, leaving the field would signal an end to editing, however, in a drawing editor, the moving of a figure may be considered complete when the user stops dragging it. Thus, the infrastructure needs to be informed of such events

but also must not be bound to specific application events. To solve this problem we define standard synchronization events that allow such information to be passed along to the infrastructure.

A synchronization event redefines the values of the transmission and correctness attributes of the property operations that are still in the buffer. Specifically, such an event is a four-tuple consisting of:

- A global identifier of the object on which the operation(s) are performed;
- The name of the property affected. This parameter could be `null`, in which case all properties of the object are implied;
- A new value for the transmission parameter (*Increment, Complete, Scheduled,* or *Transmit*);
- A new value for the correctness parameter (*Raw, Parsed, Validated,* or *Committed*).

For example,

```
<"Outline", "section", "Complete", "Raw">.
```

specifies that all property operations on the *section* property of the object named "Outline" should be relabeled as *Complete* and *Raw* unless they have higher values already. Once relabeled, all affected events must be reevaluated with respect to the current sharing policy and sent out, if necessary.

Our implementation uses synchronization events to implement the explicit *transmit* and *commit* commands, that is, whenever the user issues a transmit/commit command, a corresponding synchronization event is issued. Similarly, timer-initiated communication is also implemented by generating appropriate synchronization events. This scheme also allows for system-defined communication operations—transmit/commit—to be triggered automatically as a side effect of specific user actions, such as releasing a shape object in a drawing editor.

By default, synchronization events affecting an object are also implicitly applied to all dependent objects in the structural hierarchy. Thus, a *commit* event on a section would commit all changes in that section, while a *commit* event on the root outline object commits all changes to the whole outline.

3.3.4 N-user Sharing

So far, we have only considered the simplest case of sharing, which involves sharing of two peer objects. In general, however, we expect that more than two users may participate and may have different preferences in their collaboration. To accommodate this scenario, we extend our model to the *N*-user case, by representing it as a set of pair wise collaborations. Since every user could potentially have a different sharing policy for each of his collaborators, logically, we need to maintain separate state information regarding the sharing with each remote user. For that purpose, for each user (or group of users), we maintain a separate outgoing queue, which is governed by the current effective sharing policy with the respective user.

Figure 3.4 depicts the configuration for three users—at each replica there are two outgoing queues, which enables the infrastructure to support different sharing modes between each pair of users. For example, user *A* may share incremental updates with user *B* but only completed updates with user *C*. Thus, as *A* edits the outline, its '*B*' outgoing queue stays empty because all updates are sent out immediately. At the same time, the '*C*' outgoing queue would buffer incremental updates until it receives a proper synchronization event.

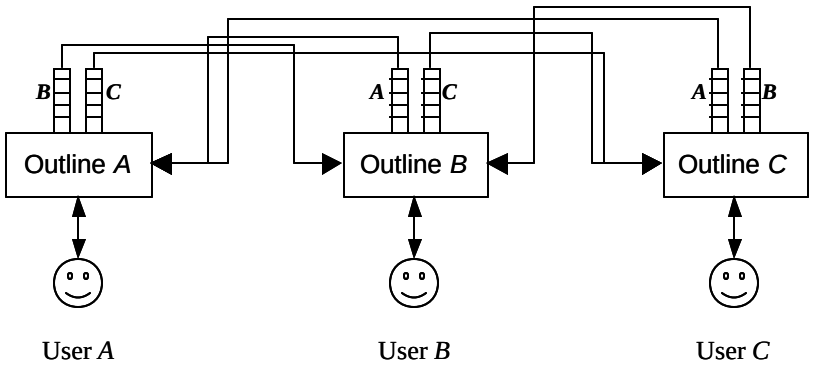


Figure 3.4 ***N-user Sharing Example***

Thus, each property and synchronization event is (logically) replicated and filed in all outgoing queues and each copy is subjected to the rules of the effective sharing policy for that queue. Also, if diff-ing is used, as shadow reference copy of the shared object reflecting the state of the object at the last diff invocation with respect to each user is maintained. Thus, the model allows the sharing parameters for each pair of users to be potentially different.

Like *Suite*, we also provide a mechanism through which users can ensure that they have identical policies—e.g., user *B* wants to have the same policies as user *A*. To implement this, we reuse our policy notification mechanism. Recall that every time a policy changes, a corresponding notification must be sent to all affected parties and the policy matching must be performed again. Thus, to ensure identical policies, instead of performing a match, we replace the policy with that of the reference user. In the above example, every time user *B* receives a policy notification from *A*, its current policy is directly replaced with the received policy.

3.3.5 Object Diff-ing

Let us now return to the generic object diff-ing service that can generate events on behalf of applications that do not implement a suitable notification mechanism. As a driving problem, consider the semantic sharing of the outline object. Since in the single-user case the application does not need to issue events in response to changes to its state, there may be no suitable notification mechanism that the developer can adapt for the multi-user case. However, performing a diff operation on a regular basis would provide most of the benefits of fine-grained shared without incurring the cost of modifying the original application.

The basic idea of object diff-ing is to look at consecutive snapshots of the state of an object and to deduce the sequence of operations that have been applied in the interim period. We should first acknowledge that this is not a problem that can be solved unambiguously. In fact, for every starting and ending state there are an infinite number of possible operations that would be consistent with the starting and ending state. However, in most situations, an educated guess will be a good enough approximation of the real operations that took place. For example, if the starting value of the outline *title* was 'Tittle' and the end value was 'Title', it is a safe bet that a 't' has been deleted. Whether the deletion occurred at position 2 or 3 is, in all likelihood, irrelevant to the users. In most situations, the chance for correct deduction increase dramatically as the time between successive *diff*s decreases. Alternatively, we could trigger a *diff* whenever a change is *known* or is likely to have occurred (e.g., end of mouse dragging in a drawing area). While the latter option is largely application-specific, its implementation is trivial because it is reduced to issuing a synchronization event.

The basic structure of the diff-ing procedure is very similar to the `ObjectWalk` procedure discussed earlier in the chapter. In this case, however, two object structures are traversed in parallel in search of differences. The `ObjectDiffer`, which is the service interface, uses property specifications to decompose objects into properties and then, for each property, it looks up and invokes the corresponding property diff handler to obtain the difference. The results are aggregated and returned to the caller. As before, the `lookup` on line 6 refers to a lookup in the `handlers` clause of the relevant property specification. The handler is implicitly instantiated, if necessary, and an object reference is returned.

```
1  ObjectDiffer( oldObject, newObject)
2    if oldObject == null
3      return newObject
4    result = null
5    for each property p_i of oldObject do
6      differ = lookup diff handler d_i for p_i
7      result = result + d_i(oldObject, newObject, p_i)
8    end
9    return result
10 end
```

The diff handlers, for their part, follow a similar hierarchical approach, shown below. A diff handler takes as arguments two object instances representing the old and the new state of the object and a property with respect to which to compare the objects. The property differ checks whether values are primitive to the procedure and, if so, directly computes their difference. In all other cases, it recursively refers the diff-ing of the retrieved property values to the global diff procedure for further processing. This also implies that the global differ must keep track of visited objects to avoid cycles in the object structure.

```
1  DiffHandler(oldObject, newObject, property)
2    reader = lookup read handler for property
3    oldValue = reader.getValue(oldObject, property)
4    newValue = reader.getValue(newObject, property)
5    if oldValue and newValue are primitive
6      result = compute diff from oldValue and newValu
7      return result
8    else
9      return ObjectDiffer(oldValue, newValue)
10 end
```

Thus, to extend the set of properties that can be handled by the infrastructure, it is sufficient to add property-specific differs and include them in the property specifications. More importantly, we can use the same property-based approach to implement other services in a modular fashion. In particular, the implementation of two other generic services—

equality testing and deep cloning—can be based on the exact same approach. The need for these services was originally motivated by our work on the object-sharing infrastructure, however, our further research has shown these to be very useful in applications not related to collaboration, such as our ongoing work on object testing described in Chapters 5 and 7.

Equality testing refers to the process of determining whether two object structures are equivalent and is implicitly used by property handlers in computing the difference (line 6 of the `DiffHandler` code above). In *Java*, all objects have an `equals` method whose implementation determines object equivalence. However, in practice, this method is rarely implemented by programmer-defined classes and the default implementation, which compares whether the two object references point to the same physical object is not helpful because we need to compare independent copies of the object structure. Furthermore, even if the `equals` method is overridden, the infrastructure cannot determine if the provided implementation is useful for diff-ing purposes (object equality may be defined in multiple ways depending on application needs). Therefore, it is sensible for the infrastructure to have an implementation it can rely on.

Deep cloning refers to the process of obtaining an independent copy of the entire object structure. Deep cloning is needed to create an independent copy of the current state of the object after a *diff* invocation. By independent we mean that changes to one of the copies does not affect the other. The copy (initially `null`) serves as a basis for comparison on the next *diff* invocation. In *Java*, object cloning is a first-class concept, and every object tagged as `Cloneable` gets a default implementation of the service. However, this implementation is a shallow version of the cloning, which does not provide independence copies. For example, cloning the standard `Vector` class produces a copy whose elements are references to the same objects as the original one. In short, as in the case of equality testing, the infrastructure has no means of determining whether a shared object implements cloning that is useful for *diff*-ing purposes.

As already stated, property-based implementation of equality testing and deep cloning can provide reliable basis for implementing *diff*-ing. Since the implementations follow the exact same template as object *diff*-ing, we omit its detailed description to avoid repetition.

3.3.6 Application Layer Model

The described property-based sharing model provides for fine-grained sharing services, however, it also raises some correctness issues. Suppose that the outline title is edited through a text field and both the text field and the outline object provide appropriate notifications to the infrastructure. An implementation of the sharing service, which does not take into account the dependency between the text field and the outline would produce incorrect results. Such a scenario is shown on Figure 3.5: as a result of user *A*'s action, a new character is inserted into the title.

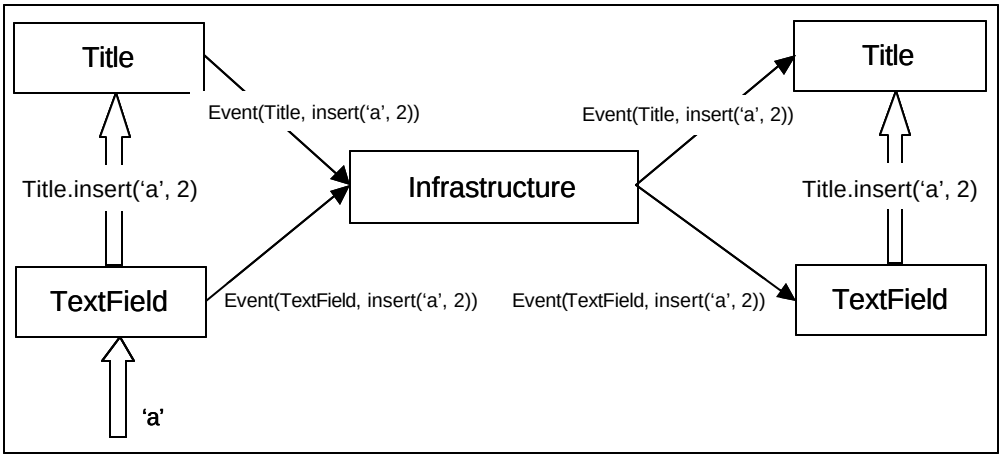
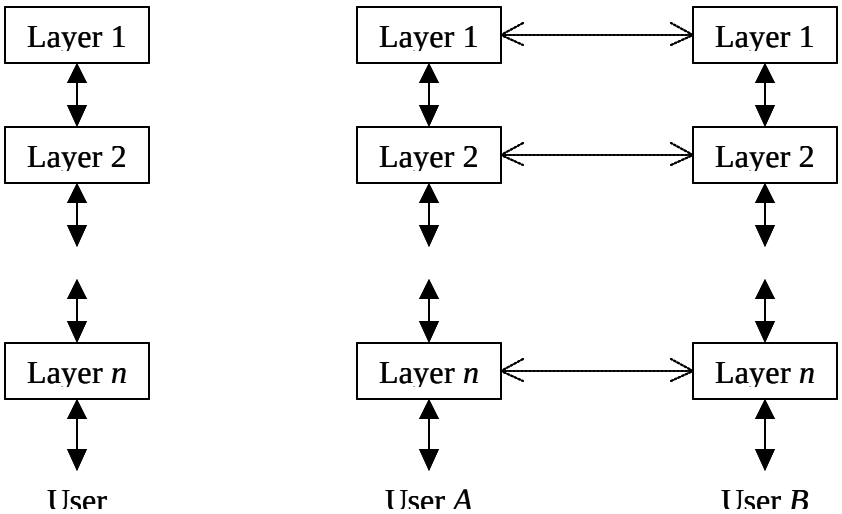


Figure 3.5 Example of multiple notification of causally related events

This results in a method invocation on the local *Outline* object and a notification event received by the infrastructure. If the infrastructure also captures and transmits the notification of the *Outline* object, this would eventually result in the peer *Outline* object receiving the original event twice, which would lead to a duplicate character insertion.

The above example is symptomatic of a more general problem that we need to address—multiple notifications of causally related events. For that purpose, we need a conceptual model that allows us to identify such events and prevent multiple notifications. Dewan's *zipper* model [7] provides a generic framework to reason about the issue. It is a generalization of the concrete layered model we used in our Related Work chapter.



a) Single-user layered architecture

b) Multi-user layered architecture: sharing may be implemented at any layer

Figure 3.6 Zipper model of multi-user applications

According to the model, a single-user application consists of a number of layers with different levels of abstraction (Figure 3.6a). At the top is the most abstract layer, which maintains the abstract state of the application. Each subsequent layer adds more syntactic details and brings the representation closer to the one seen by the user. The user interacts with the lowest layer and the relevant updates go up through the layers until they reach the top layer. In the other direction, changes to the abstract state are communicated from layer to layer and are gradually transformed into the representation seen by the user.

Given this application structure, the zipper model represents collaborative applications as two (or more) application instances whose state is shared by sharing the state of peer layers (Figure 3.6b). Returning to our multiple notification problem, we notice that causally related notifications occur as a result of user actions being translated from a less abstract to a more abstract representation, with each successive translation triggering a separate notification. The most common solution among existing infrastructures is to provide sharing at one fixed layer (e.g., shared window systems provide sharing at the window layer). The shared layer processes events received from remote replicas the same way it processes local events and propagates the results to upper (more abstract) layers, thereby achieving the sharing of those layers as well. This approach automatically eliminates the correctness problem for the application at the expense of sharing flexibility.

To increase sharing flexibility, *Suite* provides sharing at two levels (model and view) that can be dynamically switched at run-time. This is possible because the system supplies the user interface and knows the precise application layering. However, in our model, we do not have control over the user interface and, hence, we need an alternative mechanism to derive the application layering. Current programming languages do not provide any suitable mechanism to specify such architectural features so we introduce an *XML*-based layer description language for that purpose. An application layering definition consists of two parts—layer mapping and layer dependencies. Logically, the layer mapping is a set of tuples of the following form:

```
<class, property, layer>
```

The interpretation is straightforward: the specified *property* of all object instances of the given class belong to the given *layer*. Since giving a definition for each individual property would be prohibitively costly, we support two rules for supplying default layer assignments. First, the *property* could be `null`, in which case all properties in the class are implied. Second, if no explicit definition is found for a particular class, we recursively lookup the definitions for the superclasses until an appropriate one is found. Following the class-inheritance hierarchy can be very efficient: for example, all UI widgets (windows, icons, menus, etc.) in *Java* descent from the `java.awt.Component` class. Thus, with a one-line definition—`<java.awt.Component,null,"view">`—we can specify that all widgets belong to the *view* layer.

Layer dependencies describe *is-an-editor-of* relationships between any two layers, e.g., *view* → *model*. To illustrate application-layering descriptions, let us consider the example layer decomposition of our outline application shown on Figure 3.7.

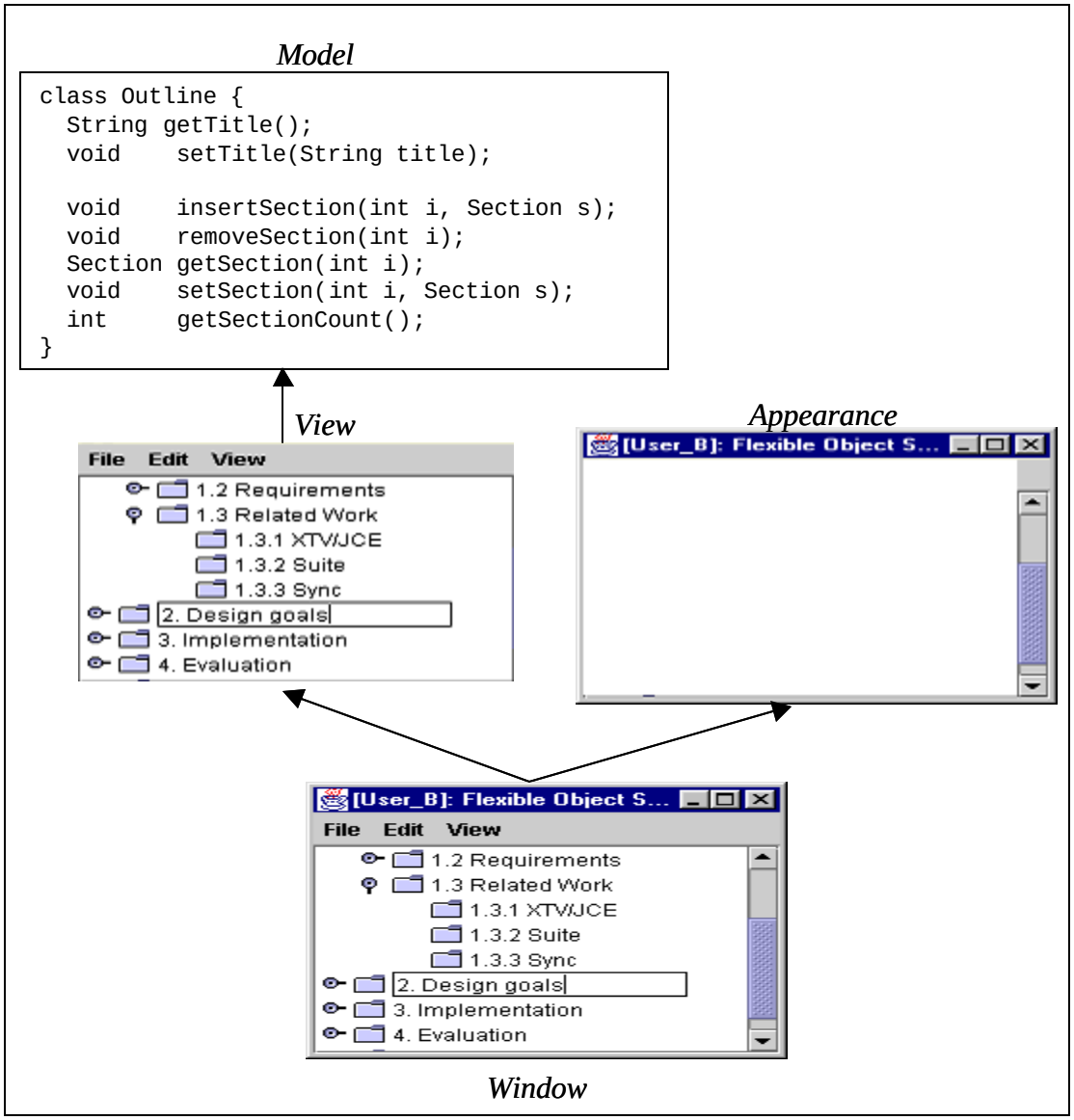


Figure 3.7 Layer decomposition for outline application

At the lowest level is the *window* layer, which consists of the single application window through which all objects are edited. The *view* layer consists of the window's menu and a `JTree` object through which the outline object is edited. The appearance layer consists of the elements of the application window that do not affect the state of the outline, such as the scrollbar. In this example, a user action may trigger one of two sequences of events. If the user performs an action that modifies the outline, the process triggers three causally related notifications at the *window*, *view*, and *model* layers, respectively. Alternatively, if a user

action (e.g., dragging the scrollbar) concerns only the *appearance* layer, it triggers a sequence of two causally related notifications—at the *window* and at the *appearance* layers.

Thus, if *window* sharing is specified, all notifications from other layers must be suppressed. If *view* sharing is specified, then *model* and *window* notifications must be suppressed. Similarly, if *model* sharing is specified, *window/view* events are suppressed. Since the *appearance* layer is independent of both the *model* and *view* layers, its sharing can be turned on/off independently of the *model* and the *view*.

In addition to correctness, this generic layer model also gives us a high-level mechanism for sharing specification by dynamically changing the shared layer. In other words, it enables us to dynamically "open" and "close" the zipper. Opening is in fact a trivial task because we move from tighter sharing modes to more relaxed ones and all we really need to do is to switch from sharing *window* events to sharing *model/appearance* events. However, the reverse process is non-trivial in the general case. To illustrate this, consider the following scenario: Initially, users are using asynchronous sharing of the model, which means that they may have completely different versions of the shared object. If they want to switch to *view* sharing, the infrastructure must first bring the outline model versions into a consistent state. To switch to *window* sharing, both the *view* and the *appearance* must be consistent beforehand.

In general, to switch the sharing from a higher (more abstract) layer L_k to a lower layer L_m, the infrastructure must ensure that all layers that depend on L_m are brought into consistency first. Note that the above transition process must be followed consistently step by step starting with layer L_k as described in the above example. We could not, for example, take the user interface of replica *A* of the outline, clone it and attach it to replica *B*. Conceptually, the problem is that the connections between the semantic object and its user interface are inherently application-specific and, hence, recreating them cannot be done without input from the programmer. In practical terms, cloning user interface objects in *Java*, for example, does not yield any usable results because part of the internal state of the standard user interface objects is related to the native implementation of the window system.

From the above discussion we conclude that, a layer description mechanism, such as ours, is necessary in order to implement dynamic switching of the shared layer.

3.4 Sharing Specification Model

Our basic approach of using properties as the fundamental unit of sharing implies that all collaboration policy specifications will have to be expressed at the property level. Given that the number of properties can easily run into the hundreds even for applications of modest size, the specification effort would be too costly to be practical. Therefore, we employ an inheritance-based specification model, similar to that of *Suite*, layer-based macro commands, and policy naming to considerably lower the cost of specification. As the following discussion shows, the main conceptual difference between our inheritance model and that of *Suite* is in the way we define the inheritance hierarchies and, in particular, the structural hierarchy of shared objects. In *Suite*, it is based on an interpretation of the concrete implementation of the shared objects, whereas in ours it is based on the observable logical dependencies among these objects. Once the hierarchies are established, however, we follow the same rules to find suitable sharing policies as *Suite* does.

3.4.1 Inheritance-Based Specification

Like in *Suite*, the basic idea behind inheritance model is to employ the existing, programmer-defined hierarchies among shared objects to provide default sharing parameters at various levels of granularity. The basic idea is that parameters for objects higher in the hierarchy provide default values for those that are below.

In our model, we utilize two kinds of hierarchies: structure-based and type-based. Structure-based (or structural) hierarchies arise as a result of Has-A parent-child dependencies within application structures. In our case, we use the set of *observable* object properties to determine such relationships. Thus, the *outline* Has-A *title* property and Has-A *section* property. In turn, the *title* Has-A value and the *section* Has-A set of values and the analysis is carried out recursively on the corresponding property values. Thus, the *title* property, having a string value, defines a single child, whereas the *section* property potentially defines multiple children—one for each section. In practical terms, to find a default parameter value, we use the inverse, Is-Part-Of relationship, as we need to traverse the hierarchy bottom-up. Since objects are dynamically created and destroyed, their corresponding Is-Part-Of relationships also change dynamically. Unlike other infrastructures, such as *JCE* and *Suite*, we do not use object names to store structural hierarchical

information (e.g., Section 1.2 may have an identifier such as 'Outline.Section[1].Section[2]'). Instead, we generate object names from a flat namespace and maintain a table of structural dependencies using information from property events (implementation details are given in Section 4.2.2). This approach was chosen to avoid the need of having to rename (potentially large parts) of the shared structures whenever structural changes are introduced and to avoid parsing the names to locate a target object. Our experience shows that the associated overhead does not present a performance issue.

Type-based hierarchies arise as a result of IS-A child-parent dependencies among application objects. For example, a section instance IS-A `Section`, which IS-A `Object`. Since such relationships are defined at compile time and remain static, there is no need for our infrastructure to maintain any additional information.

As already pointed out in our discussion on *Suite*, inheriting parameters along the structural and type-based hierarchies is appropriate in different scenarios that, generally, compliment each other. However, they also create ambiguity, e.g., if no parameter is given for Section 1.1 should we look up to the structural parent (the *section* property of the outline object) or the type-based parent (the `Section` class) for guidance. We associate an optional Suite-like *inheritance directive* parameter, which points to the desired direction of resolution and has the following values:

- *structure-first*: traverse up the structural hierarchy; upon failure, traverse the type-based one;
- *structure-only:* traverse up the structural hierarchy **only**.
- *type-first*: traverse up the type-based hierarchy, upon failure, traverse the type-based one;
- *structure-only:* traverse up the type-based hierarchy **only**.

By default, we use the *structure-first* option. The rationale here is that the structural hierarchy is closer to the user's perception of the hierarchical relationships within the shared object (we also display it in a separate window). The type-based hierarchy implies some knowledge of the implementation classes and is more suitable for specification by a programmer/administrator and as a backup to user-specified parameters.

3.4.2 Policy Naming

Our model provides for sharing policies to be given names and stored persistently, which allows, for example, a more experienced person to define a number of appropriately named policies that can be used by less experienced ones. In other words, this allows users to reuse policies they have found useful in previous collaborations. It also facilitates their social coordination when using out-of-band communication, such as a live audio connection or a chat session, by providing a common name to refer to specific policies.

Also, the infrastructure uses naming to locate policies of last resort: if the hierarchical lookup described above fails, the infrastructure looks up a "default" policy, which comes with the infrastructure and must always be present. In all cases, the user has the last say and can either select a new named policy or bring up a policy editor and fine-tune the policy to his liking.

3.4.3 Macro Command Specifications

Another mechanism we provide to simplify the specification process are *layer macro commands* to manipulate the sharing parameters of entire application layers. Unlike the inheritance approach, the automation is not based on providing default values but on setting concrete values for each object (or property) of an entire application layer. The motivation behind macros is simple—if the users are satisfied with the basic layer sharing scenarios, e.g., commit-based model sharing, they should be able to directly specify it. Depending on the application design, however, this may not be readily achievable under the inheritance schemes.

One problem comes from the fact that application object structures often follow more complicated patterns that the clear parent-child relationships we assumed so far. For example, a *model* object and its corresponding *view* object often keep observable references to each other. Thus, depending on the starting point of our structural analysis, we may conclude either one of the two objects to be the structural parent of the other. Given that object structures change at run time, shifting structural inheritance may lead to user confusion, or undesirable results. The type-based approach, in addition to being more difficult for end-users, may also be unsuitable. Application objects sometimes mix functions typical of different layers by inheriting from a class that belongs to a different layer. Thus,

objects belonging to the same layer would not have a suitable common predecessor that would allow an easy type-based specification of layer sharing.

In contrast, layer macros offer a simple mechanism that is easy to understand—the user points to a layer in the layer tree and selects a sharing policy. Returning to our problem of specifying commit-based sharing for model objects, the user can simply bring up the layer tree, point to the *model* layer and select (or create) a commit-based policy. Based on the layer descriptions, the infrastructure automatically adjusts the policy for objects/properties in the layer. The macro operation, while potentially expensive for large applications, should be a relatively infrequent event and is unlikely to adversely affect the collaboration.

An alternative approach to implement layer-based specification is, again, through inheritance by employing the Is-Part-Of relationship between objects and layers to define the hierarchy. However, we have made the conscious choice not to go along this path because we believe that the process of figuring out the exact sequence in which the three hierarchies (structural, type, and layer-based) would be traversed and what would the outcome be introduces too much complexity and confusion for the user.

3.5 Summary

In this chapter, we described the conceptual basis upon which our infrastructure implementation is built. We introduced a shared abstraction model based on programming patterns that enables the infrastructure to automatically derive the logical structure of a larger class of abstractions than currently possible. We showed that, as a means of describing abstractions, programming interfaces and *JavaBeans* are special cases of the use of patterns.

We introduced a new architectural model based on property handlers that separates application-specific and collaboration concerns and, thereby, facilitates extensibility, automation and code reuse.

We also presented a new sharing model that permits flexible parameter-controlled object sharing based on properties. The set of supported parameters provides control at variable levels of granularity starting from the entire shared application artifact down to the individual property. Our sharing parameters a proper extension of the original *Suite* coupling parameters and allow a wider spectrum of sharing modes supported by current infrastructures to be modeled. Furthermore, we have introduced object diff-ing as generic service that allows fine-

grained asynchronous object sharing to be performed while reusing in full the single-user object implementation. We have defined a generic event protocol consisting of property and synchronization events that allows collaboration features to be gradually introduced into the application and permits application-specific events to be used as triggers for collaboration-related operations. Finally, we developed a layer specification mechanism that allows correct object sharing to be implemented at different layers of the application while allowing a dynamic switch of the shared layer.

Our specification model adapts the original *Suite* hierarchical model to allow collaboration specification based on structural inheritance and type-based inheritance for object-based programs. We have employed object properties as a means of defining and dynamically maintaining structural dependencies and have introduced macro commands as an easy way to dynamically control sharing based on application layers.

We conclude our discussion by outlining how our sharing model can handle different sharing modes supported by reference infrastructures.

3.5.1 Window Sharing

The implementation of window sharing is rather straightforward—we model it as synchronous sharing of the window event queue. The queue is modeled as a simplified version of a *sequence* property defined by the *postEvent/peekEvent* pair of pattern methods. The input events generated by the active user are captured by a specialized adapter object, translated into property events, and eventually replicated at all other participants. To achieve synchronous sharing, we choose *Log* for the acquisition parameter, <*Raw*,*Increment*> for both the outgoing and incoming processing parameters, and *Playback* for the installation. Thus, as soon as a UI event is captured, it is immediately transmitted to, and posted in the window event queues of all participants. As a side effect of the posting, the window system receives and processes a local representation of the remote event, and triggers normal application processing. In this special case, there is no need to define any property handlers because the actual processing of events is handled by the UI toolkit.

3.5.2 Latecomer Accommodation

In actual collaboration, not all users may participate from the very beginning of the session. Hence the need for a mechanism that brings latecomers up to speed. Generally, there

are two approaches—log replay and current state transfer—and our framework readily accommodates both. To achieve log replay, the users must specify *Log* for the acquisition parameter, playback for the installation parameter and a communication parameter that is higher than *Raw*, e.g., *Transmit*. Thus, whenever the new user joins, a simple *transmit* command would result in the log being sent to the latecomer. Alternatively, users could use a state transfer by specifying *Read* for the acquisition, instead of *Log*. Thus, upon a *transmit* command, the infrastructure will read the current state and send it to the newcomer. We also offer a third option, which is to use *Diff* instead of *Read* as an acquisition parameter. Thus, assuming that both the sender and the newcomer started with the same version of the shared object, e.g., loaded from a file, the *Diff* option will generate a history of updates between the base and the current version.

3.5.3 Suite

Since we used *Suite*'s sharing model as a basis for our own, we can naturally accommodate its different sharing modes by fixing the acquisition and installation parameters to *Read* and *Replay*, respectively. To simulate its model based on active and interaction entities, we use an additional, master copy of the shared object, which is not associated with any user and we set its sharing policy to *Committed*. Thus, the simulated active entity receives only committed updates, whereas interaction entities, edited by different users, can exchange intermediate results (as in *Suite*, *committed* updates are forced on all users).

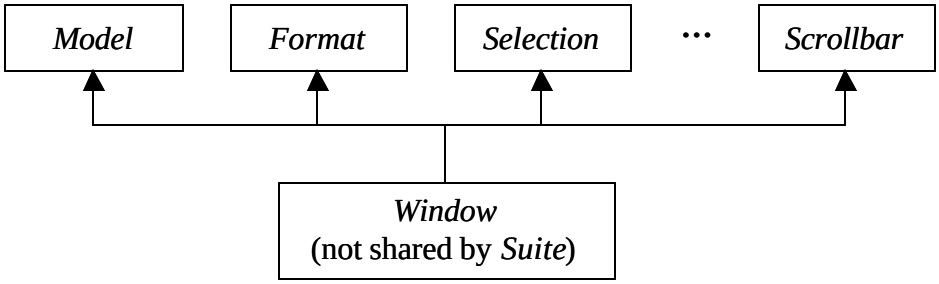


Figure 3.8 *Suite*'s Layer Model

To model *Suite*'s layering model, we map the model object, as well as the separately controllable UI coupling attributes—selection, format, window size/position, etc.—to separate layers with no dependencies among them (Figure 3.8). In other words, in *Suite*'s

case there is no issue with correctness because each user action affects exactly one layer of the application. While this approach works for the system-defined user interface, it does not generalize to the (typical) case of custom-built UI, such as the one in our Outline application.

We should also note that *Suite*'s implementation of WYSIWIS sharing is different from that of shared UI systems—it relies on sharing all the layers it defines above the window layer defined by OS as a means of presenting identical views of the application to all users. Again, this is only a limited implementation, which would only work for applications with *Suite*-generated UI.

Finally, we also support *N*-user sharing with arbitrary combinations of sharing parameters as *Suite* does.

3.5.4 Asynchronous Sharing

Asynchronous sharing, such as the one in *Sync*, can be modeled by *Log*-ing all updates and transmitting them explicitly. In turn, the response from the *Sync* server, will be sent back as a set of *Committed* updates, forcing the client to execute the corrective commands. The merge handlers will perform the actual merge of updates, which could be parameter-driven (e.g., *Sync, TACT*) or a specialized procedure (*Bayou*). Our *Diff* capability gives us the additional option of deriving the changes in case the application does not use a notification mechanism.

4. INFRASTRUCTURE IMPLEMENTATION

In the previous chapter, we presented a conceptual model for implementing a flexible object-sharing infrastructure that better satisfies our requirements. In this chapter, we motivate and explain the specific implementation choices that we have made in mapping this conceptual model to a specific implementation based on *Java* and *XML*. Later on, we use this prototype implementation in our evaluation process.

We use our outline example to walk through the implementation steps necessary to implement a multi-user version of it. In the process, we describe the details of our implementation and their relationship to the development effort.

4.1 XML Pattern Specification Language

Recall that our driving problem is to implement flexible, fine-grained sharing of the example implementation of the `Outline` object shown on Figure 3.1. For that purpose, we first need to specify the programming patterns that would permit the infrastructure to extract the object's logical components—its properties. Our general approach has been to provide an extensible framework in which property definitions can be plugged in as needed. At the same time, we need to provide a minimal set of ready-to-use patterns that directly support the sharing of the most commonly used abstractions. To determine this set, we have used previous work as guidance—the basic rationale is to be able to simulate the abstractions of existing systems, such as *Sync*. Therefore, by default, we support three basic types of object properties—*simple*, *sequence*, and *table* properties. Simple properties are identical to *JavaBeans* properties, *sequence* properties have the same semantics to *Sync*'s *ReplicatedSequence*, whereas *table* properties correspond to *Sync*'s *ReplicatedDictionary* abstraction.

In the Outline case, we need two of the definitions—the ones for simple and sequence properties. Figure 4.1 shows the actual *XML* property specification for simple properties, with the sequence (and table) definitions given in the Appendix. Although more verbose, the *XML* specification shown on Figure 4.1 is, in fact, equivalent to the one we described in the

previous chapter using a simplified syntax, with the addition of several minor features, whose purpose is explained below.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE property_spec SYSTEM "PropertySpec.dtd">
<property_spec type = "simple" version = "1.1">
  <pattern name = "getter" semantics = “accessor”>
    <return_type>
      <variable name = "GetType"/>
    </return_type>
    <method_name>
      <literal>get</literal>
      <variable name = "GetName"/>
    </method_name>
  </pattern>
  <pattern name = "setter" semantics = "modifier">
    <return_type>
      <literal>void</literal>
    </return_type>
    <method_name>
      <literal>set</literal>
      <variable name = "SetName"/>
    </method_name>
    <argument_type>
      <variable name = "SetType"/>
    </argument_type>
  </pattern>
  <constraint predicate = "equals">
    <lhs>  <!-- Left-hand side of the constraint -->
      <reference variable = "getter"/>
      <reference variable = "GetType"/>
    </lhs>
    <rhs>  <!-- Right-hand side of the constraint -->
      <reference variable = "setter"/>
      <reference variable = "SetType"/>
    </rhs>
  </constraint>
  <constraint>
    <lhs>
      <reference variable = "getter"/>
      <reference variable = "GetName"/>
    </lhs>
    <rhs>
      <reference variable = "setter"/>
      <reference variable = "SetName"/>
    </rhs>
  </constraint>
  <name_rule>
    <reference variable = "GetName"/>
  </name_rule>
  <type_rule>
    <reference variable = "GetType"/>
  </type_rule>
  <handler operation = "read"  class_name = "handlers.SimpleReader"/>
  <handler operation = "write" class_name = "handlers.SimpleWriter"/>
  <handler operation = "clone" class_name = "handlers.SimpleCloner"/>
  <handler operation = "diff"  class_name = "handlers.SimpleDiffer"/>
</property_spec>
```

Figure 4.1 *XML* specification for simple (*JavaBeans*) properties

4.1.1 Property Versions

One of the features not included in our conceptual description is the `version` attribute of the property. Its main purpose is to help resolve ambiguities in determining which property definition is used in cases where more than one specification define a property with the same name. To illustrate, consider the following *index-based* implementation of a sequence property. To avoid repetition, we omit the exact property specification but indicate as comments the names of the pattern methods used.

```
public void    insertSection(int i,Section s); // "insert" method
public void    removeSection(int i);           // "remove" method
public Section getSection(int i);              // "lookup" method
public int     getSectionCount();              // "count"  method
```

It implements the bare minimum of methods necessary to manipulate the sequence. However, a subclass, may implement an extended version featuring additional (convenience) methods, such as the ones shown below

```
public void setSection(int i,Section s);
public void removeAllSections();
```

Hence, if the developer presents two versions of the sequence—one matching the minimal pattern above and one matching the extended one—the infrastructure would discover that they both define a property named *Section* in the subclass. To resolve the ambiguity, the infrastructure picks the property specification with the higher version number.

By convention, versions that have the same major version number should only differ in their method names. In other words, they should have the same number of methods with the same signatures and semantics, which enables them to directly reuse handler code. For example, if we used a *title*/*setTitle* naming convention for our title property methods, we would need an alternative *simple* property specification, which would have the same *major* version and `handler` elements but a different minor version and `getter` method pattern specification.

In contrast, properties with different major version numbers may not be able to directly share handler code because they may have incompatible implementations. Consider the following alternative implementation of a sequence property based on a *Java*'s `Enumeration`:

```
public void insertSection(int i,Section s); // “insert” method
public void removeSection(int i);           // “remove” method
public Enumeraton getSections();            // “list” method
```

The `Enumeration` allows a single iteration over the elements of a collection of objects in the proper order. It has two methods:

```
public Object nextElement()
public boolean hasMoreElements()
```

The first one returns the next element in the enumeration (throws an exception if none), whereas the second one is a predicate, which tests whether the enumeration has more objects that have not been inspected so far.

While the index-based and enumeration-based sequence implementations are logically equivalent in that all the operations of one implementation can be emulated using the other, there is no one-to-one correspondence of individual methods and, therefore, handler code cannot be directly reused. However, the property implementation differences are transparent to the infrastructure services because handlers export the same programming interface by defining the same abstract property operations regardless of the underlying property implementation.

To illustrate, let us consider the actual implementation[4] of the object registration that is invoked by the application’s startup code to register shared objects. Essentially, the registration consists of traversing the object tree as described in our basic `ObjectWalk` algorithm from Chapter 3 and registering all `Identifiable` objects with the `CentralRegistry`. To perform this registration, the infrastructure needs a *read* handler for each discovered property so that it can perform the recursive analysis. In this example, we need a *read* handler for each of the two property versions (index- and enumeration-based).

The infrastructure is insulated by the specifics of the *read* handler implementations by requiring them to implement the `PropertyReader` interface:

PropertyReader interface:

```
import java.io.Serializable;
import colab.bus.pattern.Property;
public interface PropertyReader extends Serializable {
    public Object[] getValue(Object target, Property property);
}
```

[4] For presentation purposes some details, such as exception handling, have been omitted.

The infrastructure code (shown below) uses the introspection mechanism to discover the object properties and locate a *read* handler for each of the properties it discovers. If a new property definition is introduced, this would result in (potentially) discovering more properties, however, this will require no changes to the infrastructure or application code. The only piece of code that needs to be added is a *read* handler for the new property.

Object registration code (infrastructure service):

```
void dfsRegister( Object node, Object parent, PropertySpec[] specs) {
  if( node == null)                // Ignore null objects
      return;
  if( isPrimitive( node))          // Ignore primitive types, e.g. int
      return;
  if( visitedObjects.containsKey( node)) // Ignore visited objects
      return;
  if( node instanceof Identifiable) // Register Identifiable objects
      CentralRegistry.bind((Identifiable)node);
  visitedObjects.put( node, "");    // Add object to visited list
  String className = node.getClass().getName();
  Property[] mp = Introspector.getMatchedProperties( className, specs);
  for( int i=0; mp!=null && i<mp.length; i++) {
    PropertyReader reader = ((PropertyReader)mp[i].getHandler( “read”);
    Object[] values = reader.getValue( node, mp[i]);
    for( int j=0; values != null && j<values.length; j++)
      dfsRegister( values[j], node, specs);
  }
}
```

To complete our example, we also present concrete implementations of the *read* handlers of the two version of the sequence property.

Read handler for the *index-based* implementation:

```
import java.lang.reflect.Method;
import colab.bus.pattern.Property;
public class IndexSequencePropertyReader implements PropertyReader {
  public Object[] getValue(Object target, Property property) {
    Method countMethod = property.getMethod(“count”);
    int count = countMethod.invoke(target, null);
    Method lookupMethod = property.getMethod(“lookup”);
    Object[] values = new Object[count];
    for(int i=0; i<count; i++) {
      Object[] arg = new Object[] {new Integer(i)};
      values[i] = lookupMethod.invoke(target, arg});
    }
    return values;
  }
}
```

Read handler for *enumeration-based* implementation:

```
import java.lang.reflect.Method;
import java.util.Enumeration;
import java.util.Vector;
import colab.bus.pattern.Property;
public class EnumSequencePropertyReader implements PropertyReader {
  public Object[] getValue(Object target, Property property) {
    Vector seq = new Vector();
    Method listMethod = property.getMethod("list");
    Enumeration list = listMethod.invoke(target, null);
    while(list.hasMoreElements())
      seq.addElement(list.nextElement);
    Object[] values = seq.toArray();
    return values;
  }
}
```

4.1.2 Method Semantics Attribute

Another important attribute we have not discussed so far is the `semantics` of a method. It provides a basic classification of the method's functionality, which is necessary to correctly implement sharing and to maintain the registration information about the shared objects. We distinguish among four types of methods—*accessors*, *constructors*, *modifiers*, and *destructors*. *Accessor* methods, such as the *getter* in simple properties, are used to obtain information about the state of an object without modifying it. *Constructors*, such as the *insert* method in sequence properties, structurally change the property by expanding it with additional elements. *Destructors*, such as the *remove* method in sequence properties, are the opposite of constructors—they reduce the property structure by removing elements. *Modifiers*, such as the *setter* in simple properties, update the property but do not introduce structural changes. In Section 4.2.2 we describe how these values are used to dynamically maintain the correctness of object registration information.

4.1.3 Advanced Naming Conventions

The *XML* schema also enables the description of more complex, but common, programming patterns than the ones discussed so far. Consider the following three methods, part of the standard `java.awt.Window` class, which we would like to represent as a version of a sequence property, called *components*:

```
public java.awt.Component add(java.awt.Component comp);
public void remove(java.awt.Component comp);
```

```
public java.awt.Component[] getComponents();
```

To accommodate such cases, we provide three additional tags to be used in the `constraint` clauses:

- `short`—denotes the abbreviated version of the class name after removing the package prefix, e.g., `short(java.awt.Component) == Component`;
- `plural`—denotes the plural form of a noun, e.g., `plural(Component) == Components`;
- `plural_short`—denotes the combination of the above two, e.g., `plural_short(java.awt.Component) == Components`.

Using these tags, it is fairly straightforward to create a version of the sequence property specification that accommodates the above case—the exact *XML* code is given the Appendix.

4.1.4 Property Exclusions

In practice, property definitions must allow for exceptions to the basic rules by enabling the exclusion of a property that conforms to the syntax but not the assumed semantics of a particular property. As an exclusion example, consider the following actual piece of the implementation of the standard `java.awt.Component` class (the superclass for all user interface objects):

```
public class Component {
  public Rectangle getBounds() {
     return new Rectangle(x, y, width, height);
  }
  public void setBounds(Rectangle r) { …}
  ...
}
```

Apparently, the pair of `getBounds/setBounds` methods defines a simple *Bounds* property. Thus, during the property analysis, the `getBounds` method would be invoked to obtain the current value of the property. Following the recursive property analysis, the resulting `Rectangle` would also be analyzed. Unfortunately, the `Rectangle` also has a *bounds* property, which is implemented in a similar fashion:

```
public class Rectangle {
  public Rectangle getBounds() {
     return new Rectangle(x, y, width, height);
  }
  public void setBounds(Rectangle r) { …}
  ...
}
```

Further property analysis of the `Rectangle` following the `ObjectWalk` algorithm from Section 3.2.1 (essentially, a depth-first traversal) would result in an infinite recursion because, as a side effect of the invocation of the `getter` pattern method, the object structure would grow infinitely. Since new objects would be generated at every step of the analysis, the standard mechanism of keeping track of visited objects to avoid cycles would be of no help to break the infinite recursion (newly generated objects would never be on the *visited* list).

Generally, it is hard envisage a scheme that would automatically detect and get around such problems. Our solution is to add an `exclude` tag to the property definitions that would detect such a property, thereby directing the infrastructure to ignore it:

```
<exclude class_name="java.awt.Rectangle" name="Bounds" scope="subclass"/>
```

The `scope` attribute here specifies that, unless otherwise marked, `Component` subclasses would also inherit the exclusion. Note that exclusion is only necessary if it not possible to observe the current state of a property (through the *read* handler) without modifying it. Thus, recursive properties (ones in which the *read* handler returns the same type of object as the analyzed object) are perfectly legitimate. For example, if in the following implementation the `Section` class `getSubsection` and `getSubsectionCount` methods necessary to obtain the current state do not modify the state of the `Section` object (or create new objects), the property analysis can proceed without problems.

```
public class Section {
  public Section getSubsection(int i);
  public void    setSubsection(int i,Section s);
  public int     getSubsectionCount();
  public void    insertSubsection(int i,Section s);
  public void    removeSubsection(int i);
  …
}
```

4.2 Object Naming and Registration

So far we dealt with the implementation of the logical decomposition of shared objects. With that step accomplished, we are ready to look into the implementation of the run-time sharing mechanism. The first issue we need to resolve is *what* is being shared. For two, or more, distributed object replicas to be shared, they must have a common, globally unique identifier (or name), which allows the sharing mechanism to match the replicas and to distinguish them from all other objects. Since the issue of global naming does not arise in the

single-user case, we face the choice of implementing an implicit naming mechanism, which handles the issue transparently, or an explicit mechanism, which can take naming information from the programmer. While the implicit approach implies more automation, it also provides less flexibility in the range of sharing modes that can be supported. In particular, if two replicas are created concurrently and independently, their system-generated global identifiers would be different and the infrastructure would have no means to establish that they correspond to each other. As a simple example, suppose that users *A* and *B* have each started an instance of the Outline application. This would automatically initialize a number of shareable objects—the root `Outline` object and its user interface—each with their own unique global identifiers. However, if *A* and *B* want to share (some of) these objects, then corresponding replicas must have the same global identifier. Since the infrastructure has no information to establish the correspondence, and the programmer has no mechanism to provide it, the infrastructure must either seek help from the user, or place restrictions on the replica instantiation process (e.g., simultaneous instantiation at all sites) to ensure consistent naming. Implicitly, restrictions on replica instantiation impose restrictions on the supported sharing modes (e.g, simultaneous instantiation precludes asynchronous sharing).

Therefore, we have made the design decision to take an explicit approach to naming by making the global name part of the logical structure of each shareable object. This gives us the flexibility of getting help from the programmer in establishing a naming scheme. In addition to being more flexible, this also simplifies the translation from a local object reference to a global identifier and bypasses a registry lookup. Another benign side effect of this design is that it simplifies the management of the registry used to translate between local and global names because the global name is serialized along with the rest of the object, saving the infrastructure the need to keep track of what registry information needs to be disseminated along with every event, or the need to maintain a global registry.

We should note that, an explicit scheme becomes, for all practical purposes, implicit if the burden of assigning names is placed entirely on the infrastructure. Thus, in sharing scenarios where this is possible, the infrastructure can take charge and provide the benefits of automation typical of implicit schemes. Finally, as the following discussion shows, the additional development effort required is trivial and can be largely automated for *Java* programs.

4.2.1 Identifiable Objects

Specifically, to implement our naming scheme, we require that all shareable objects implement the following interface:

```
public interface Identifiable {
    public GID getGID();
    public void setGID(GID gid);
}
```

where `GID` is defined as follows:

```
public interface GID extends java.io.Serializable {}
```

The infrastructure provides default string-based and integer-based implementations of the `GID` interface. In addition, the system supplies a `ReplicatedObject` class, whose main purpose is to provide a default implementation of the `Identifiable` interface and to be sub-classed by application objects in lieu of sub-classing `java.lang.Object` (the root of the *Java* object hierarchy). Ultimately, the transition from sub-classing `Object` to sub-classing `ReplicatedObject` can be completely automated even for compiled *Java* code by performing it at load time by a specialized class loader using an object-instrumentation tool such as JOIE [5]. Currently, this technique is not part of our prototype as it does not bring anything conceptually new to our approach. However, we do plan to eventually incorporate it as part of our prototype to provide a more complete implementation.

4.2.2 Object Registration

The translation from global to a local reference requires that the infrastructure maintain a table, or *registry*, which maps global to local references. We refer to the process of establishing such mapping as (*object*) *registration*, and we store all such information in a `CentralRegistry` object. Another aspect of the registration is also the discovery of parent-child structural dependencies, which we store in a `DependencyRegistry` object

Object registration is triggered in one of three ways: explicitly, as a result of the application's request to `bind` an object to a specific `GID`; implicitly, as a result of recursive registration, or as a side effect of a property operation. Explicit registration is typically necessary for the root object of a shared object structure, such as the `Outline` object where the application must specify a 'well-known' name. However, in a situation where the infrastructure controls the instantiation (and, therefore the naming) of shared objects, this process can also be automated. For example, in our shared window implementation, the

infrastructure automatically ensures that all window replicas and their components have matching names. This is achieved by listening for `WINDOW_OPEN` events and automatically renaming the newly created `Window` object so that all replicas have the same name. The naming scheme is based on a counter that is consistently advanced across all application instances.

The registration of shared object structures typically follows a recursive fashion. This has two advantages: first, it frees the developer from the need to invent a naming scheme for all shared objects; second, it allows the discovery of parent-child relationships that define the structural hierarchy of the shared objects. As already discussed, our infrastructure maintains the inverse, *Is-Part-Of*, relationship.

Once discovered, structural dependencies must be maintained up-to-date with the run-time changes to the structure as a result of property operations. For example, if a subsection 1.2 is moved to section 2 as subsection 2.5, obviously, the structural dependencies would have to be updated. Thus, as a side effect of a property method, we may also need to update the dependency registry. For that purpose, we employ the `semantics` attribute of the invoked method as follows:

- If the executed method is an *accessor*, no update is necessary.
- Otherwise, if the list of parameters contains a single `Identifiable` object (e.g., a `Section`), we assume that the property operation manipulates the association between the parameter object and the target object whose property is being modified (e.g., `Outline`). Thus, we interpret the operations as follows:
  - If the executed method is a *constructor*, then a new association between the argument and the target is established.
  - If the executed method is a *destructor*, then we remove the association between the argument and the target.
  - If the executed method is a *modifier*, then we replace the current association for the argument.
- If the list of parameters contains more than one `Identifiable` object, it is unclear what association is being manipulated. Therefore, we take a conservative approach and recursively rebuild the property dependencies starting with the affected property.

4.3 Event Flow

After resolving the object naming issue, we can follow step-by-step the processing of user updates from one replica to another. Our discussion is illustrated by Figure 4.2, which gives a graphic presentation of the event flows generated by our conceptual model and summarizes possible ways in which individual updates can be shared between two, or more, replicas.

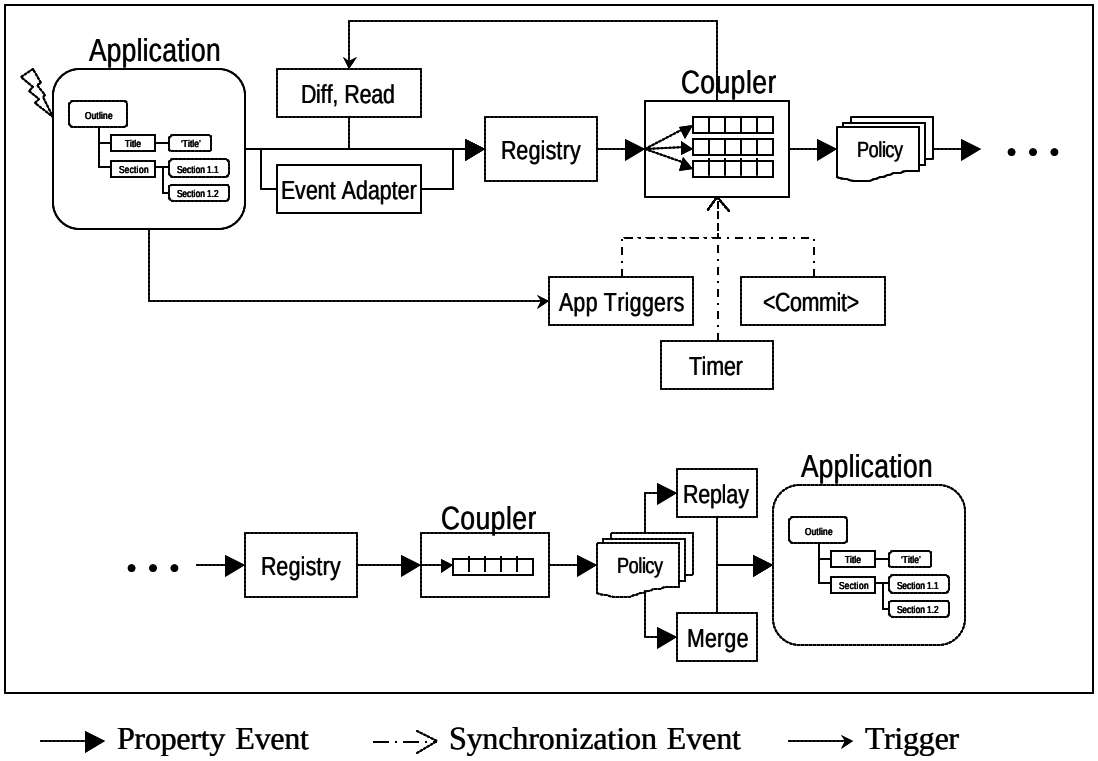


Figure 4.2 Event flow model of the infrastructure

As a starting point, we assume that the pattern analysis of the shared object (as suggested by the tree structure in the "Application" box) has been performed. Likewise, we assume that initial object naming and registration have also been completed and as a result of user actions, the local copy of the shared object is modified and a corresponding modification of the peer replicas must eventually be performed.

As a first step, we need an update notification to be issued by the application. This can happen either directly, or indirectly. By directly we mean that the application creates an instance of the infrastructure-defined `PropertyOperation` object and passes it to the

infrastructure. This approach is typical of collaborative applications that are built from scratch.

Indirect notification is typical for applications with fully developed single-user functionality and a custom event mechanism. Usually, tapping into custom event mechanism and providing an event adapter, which translates the application-specific events into `PropertyOperations` is the most economical way of interfacing the application with the infrastructure. Currently, our implementation provides two reusable event adapters—`AWTEventAdapter` and `BeanEventAdapter`. The former translates the standard UI events issued by the *AWT/Swing* library, whereas the latter translates the standard `PropertyChangeEvents` issued by *JavaBeans* objects into `PropertyOperations`.

An alternative, "pull-based", means of obtaining property events is to perform a *Diff*, or *Read* operation on (part of) the shared structure. Such an operation is invoked by the *Coupler* in response to a synchronization event whenever the current *acquisition* policy parameter is set to *Diff/Read*. This is useful when the application has not been coded to "push" events to the environment.

Synchronization events are issued as result of user action (e.g., *commit*), timer expirations, or *application trigger* activations. An application trigger is an object that listens for application-specific events (e.g., mouse release in a drawing area) and issues corresponding synchronization events (e.g., edit complete).

In all cases, events are delivered through a static, 'well-known' object, thereby bypassing the need for explicit composition between the application objects generating the event and the `Coupler` object that eventually receives them. Once the `Coupler` receives an event, its first job is to update the registry information (for that reason the diagram shows the event going through the Registry first). There are two registries that may need to be updated—the `CentralRegistry` and the `DependencyTable`. The update is based on an analysis of the `semantics` attribute of the operation and its accessor/ constructor/ destructor/ modifier classification. The `DependencyTable` is refreshed based on the rules already described in Section 4.2.2. The `CentralRegistry` is refreshed in a similar fashion—any `Identifiable` arguments of a *constructor/modifier* operations are added to the registry, whereas any `GID` arguments of a *destructor* are removed.

Next, the event is filed with each of the outgoing queues associated with individual users (or groups of users) and is then evaluated with respect to the each of the corresponding policies based on the associated user and the target object of the property operation. Events that meet the minimum requirements set by the policy are immediately sent to their respective recipients using the event multicast service described below in Section 4.5.

After the remote site receives the incoming event, it is processed along the same lines as outgoing events. First, the local registry and dependency tables are updated, then the local `Coupler` looks up the installation policy, selects the installation method, and applies the updates to the application object.

4.4 User Interface

To control the sharing of application objects, the users need an appropriate interface that allows them to control the mapping of shared objects to sharing policies, fine-tune the policies themselves, as well as execute *Transmit*/*Commit* operations. Such an interface must be generic and flexible enough to accommodate arbitrary applications and avoid the cost of building of specific interfaces for each application. Moreover, from a user's point of view, having a unified interface also allows the transfer of collaboration experience from one application to another.

4.4.1 Object Browser

The user interface of our infrastructure is built around the idea of an *object browser*. The `ObjectBrowser` shows, in a dedicated window, a tree representation of the structural hierarchy of the shared objects registered with the infrastructure (Figure 4.3).

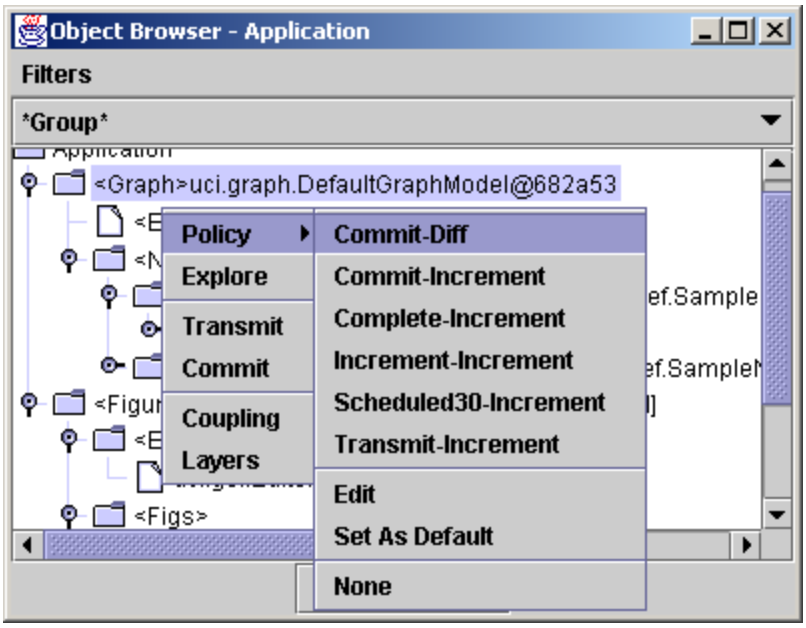


Figure 4.3 Object Browser with Policy Selection Pop-up Menu

The browser provides a generic interface that allows users to navigate the tree and execute operations on the selected objects. The operations are, in fact, callback methods registered with the browser during startup. Usually, they are infrastructure-provided, such as setting policies and committing updates. As Figure 4.4 shows, one of the operations registered is the "explore" command implemented by the browser itself. It creates an independent view (in a separate window) of the selected object and its children. The application can also take advantage of the browser and register its own operations that require a custom implementation (e.g., save, print, etc.)

The operations are triggered by the user through buttons, menu items, and context-sensitive pop-up menus. The user-selected object becomes an implicit argument in callback invocations. Furthermore, callbacks can register lists of possible values for a second argument, which are presented to the user as submenus. For example, in setting a sharing policy, the policy manager needs two arguments—the object to which the policy applies and the policy itself. As shown on the figure above, specification of these parameters can be accomplished by the user in a couple of mouse clicks: a right-click on the object and a selection of the policy from the list of named policies in the submenu.

To connect the `ObjectBrowser` with the application user interface, the application invokes (in the startup code) a registration call, which automatically adds a “Collaboration” menu to the specified application window:

```
ColabJMenu.addColabMenu(mainWindow);
```

The collaboration menu allows the user to show/hide the browser, as well as to execute general commands that do not require target selection, such as committing/transmitting all changes (Figure 4.4).

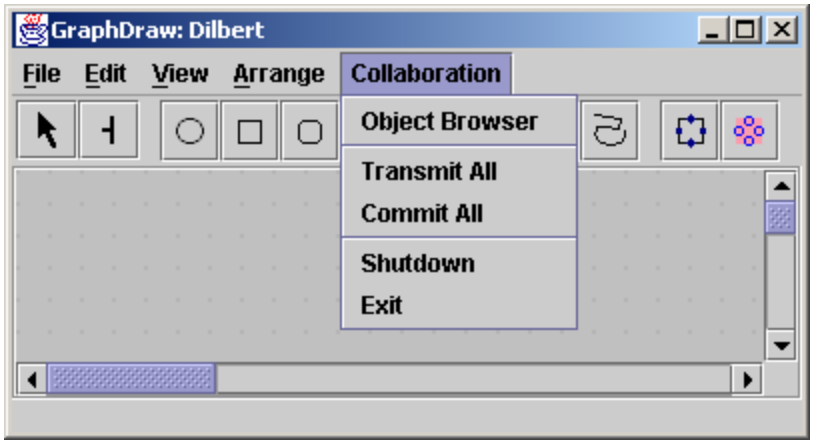


Figure 4.4 Collaboration Menu Attached to an Application

One practical issue that arises for most applications that the sheer number of properties discovered by the analysis leads to a large object tree, which may be difficult to navigate. To reduce the clutter, we offer users the option of filtering out properties based on their type or layer (Figure 4.5). In our experience, filtering out simple properties, for example, is typically sufficient to reduce the tree to a manageable size. We should note that the filtering is only for visualization purposes and does not affect the behavior of the system. Users may open several browser windows and have different filters for each one of them.

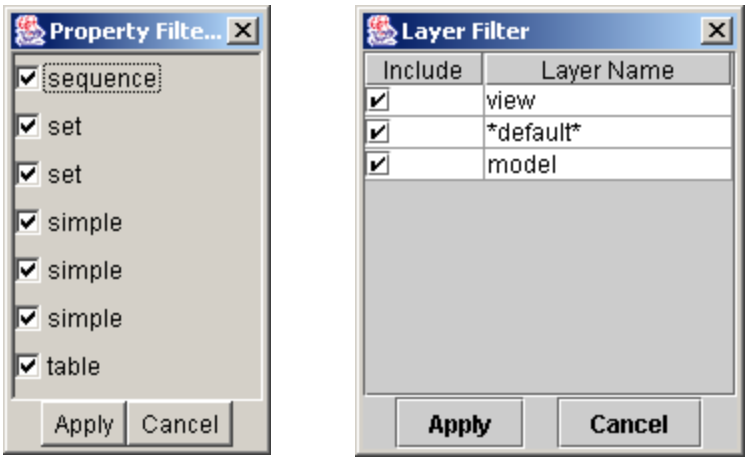


Figure 4.5 Type- and Layer-based Property Filtering

Another practical issue is related to user orientation in the application structures. In particular, how does the user map the selected node in the browser window to the actual object in the application window? Short of integrating the application and infrastructure interfaces in a custom-built interface, the best solution is to highlight in the application the object selected in the browser window. We can automatically perform this (on a limited basis) for the standard UI components, such as text fields, buttons, and panels (the infrastructure simply checks if the selected object is a subclass of `java.awt.Component` and, if so, swaps the foreground and background colors). However we cannot automatically solve the problem for non-UI objects without help from the programmer.

Currently, there are two ways for the programmer to pass on such information—by naming convention and through a "highlight" callback routine. In the first case, the developer must explicitly name the user interface component that renders the non-UI object by adding a ".view" suffix to its global identifier. In other words, if the global name of the non-UI object is `"xyz"`, then its rendering object must be named `"xyz.view"`. When the object named `xyz` is selected in the object browser, the infrastructure automatically looks for a registered `xyz.view` object and, if it finds it, swaps the values of its foreground and background properties (if present). Note that the developer does not have to generate the `GID` for the non-UI object.

The second, more flexible but less automated solution, is for the developer to register a "highlight" callback routine that changes the rendering of the selected object in the application window whenever the underlying object is selected/deselected in the browser.

Once registered, the routine is automatically invoked whenever the selection changes and the new selection is passed on as an argument. (Our name-based scheme actually uses this mechanism by registering an appropriate callback during initialization.)

4.4.2 Policy Specification

As suggested by Figure 4.3, one of the main purposes of the browser to provide a point-and-click interface for specifying both *persistent* and *dynamic*, or *custom*, sharing policies. Persistent policies are maintained in the form of *XML* files and are automatically loaded at startup from a default directory and shown to the user as a submenu. Thus, the list of available policies can be controlled by adding/removing the corresponding *XML* files and the policies can be tailored using generic *XML* editing tools. An example policy, called *Increment-Increment*, which allows all (incremental) changes to be sent and received without restriction, is specified below.

```
<?xml version="1.0" encoding="UTF-8" ?>
  <object javaClass="colab.bus.coupling.CouplingPolicy">
    <name value="Increment-Increment" />
    <transmitEvent value="0" />
    <transmitCorrectness value="0" />
    <transmitPeriod value="0" />
    <transmitTime value="0" />
    <transmitMethod value="0" />
    <listenEvent value="0" />
    <listenCorrectness value="0" />
    <listenPeriod value="0" />
    <listenTime value="0" />
    <installMethod value="0" />
  </object>
```

The main conceptual difference between custom and persistent policies is that the former are referred to by value, whereas the latter are referred to by name. Thus, naming provides a level of indirection, which simplifies the policy management and lessens the need for novice users to understand the full capabilities of the sharing model before using it. On the other hand, the option of specifying custom policies gives users complete control of the sharing process at the finest granularity

To specify a custom policy, the user performs a right-click on the target object (in the object browser) and selects “Policy” → “Edit” option shown on Figure 4.3. This brings up the coupling policy editor shown on Figure 4.6.

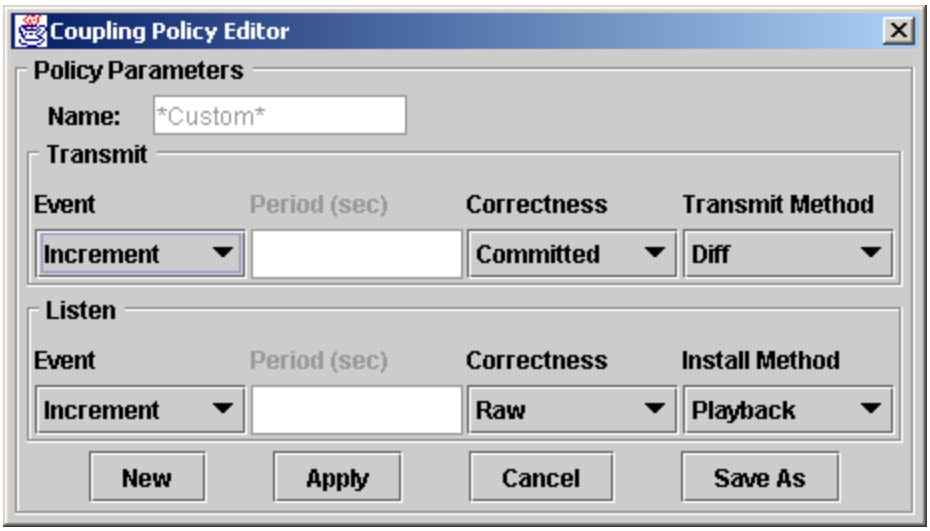


Figure 4.6 Coupling Policy Editor

If the target object/property already has a specified policy, it will be displayed as a starting point of the editing process. After editing is complete, the user has the choice of *Apply*-ing it to the target object/property, or *Save*-ing it as a persistent policy .

4.4.3 Layer Specification

Specifying the application layers is very similar to the policy specification above in that all specifications are kept in *XML* format and can also be edited through an infrastructure-provided editor. *XML* files are linked to the classes by a simple naming convention—the specification file has the same name as the class with an '.xml' extension. For example, assuming that our `Outline` class is part of the `outline` package, the corresponding layer definition would look as follows:

outline.Outline.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<object class = "outline.Outline">
  <layer name="model">
    <Title />
    <Section />
  </layer>
</object>
```

Since, in this case, all object properties belong to the same layer, we could have also used a shorthand `<ALL>` tag to indicate that fact:

```
<?xml version="1.0" encoding="UTF-8"?>
<object class = "outline.Outline">
  <layer name="model">
    <ALL />
  </layer>
</object>
```

Recall that a layer definition for a class also provides default layer assignments for properties of its subclasses. Hence, whenever a new class is encountered, the infrastructure may have to traverse several levels in the class hierarchy to find an appropriate definition. To speed up this process, we load all available layer definitions at startup time to make them available for quick table lookup.

As with sharing policies, we also provide an editor that frees the users from the low-level details of the *XML* definitions:

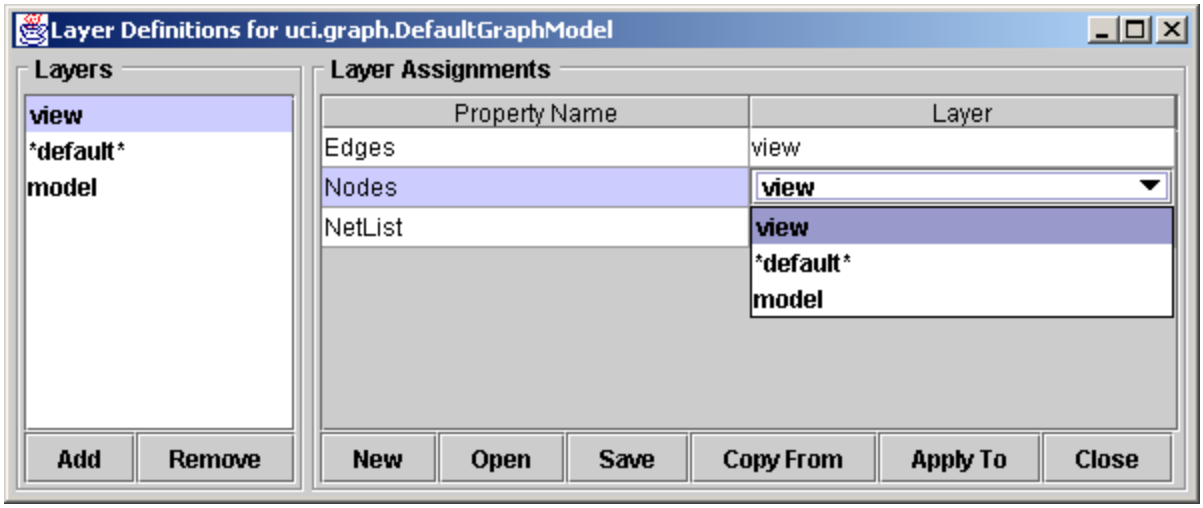


Figure 4.7 Layer Editor

Finally, we need a way of specifying the layer dependencies that are essential for the correctness of our sharing model. This is implemented via a *LayerDependencies.xml* file, which defines layer relationships specific for a particular application. For example, the layer dependencies for our *Outline* application are succinctly defined as follows:

```
<?xml version="1.0" encoding="UTF-8" ?>
<layer_dependencies>
  <window>
    <view>
      <model/>
    </view>
    <appearance/>
  </window>
</layer_dependencies>
```

We should note that layer dependencies do not have to be a strict tree structure. For example, if we attached a chat tool to the outline editor the layer dependency structure could look as a forest with the editing and chat parts having separate dependency trees (assuming they run in separate windows):

```
<?xml version="1.0" encoding="UTF-8" ?>
<layer_dependencies>
  <outline_window>
    <outline_view>
      <outline_model/>
    </view>
    <outline_appearance/>
  </outline_window>
  <chat_window>
    <chat_view>
      <chat_model/>
    </view>
    <chat_appearance/>
  </chat_window>
</layer_dependencies>
```

4.5 Remote Communication

As a matter of design choice, we have not bound the implementation of our sharing mechanisms to a particular remote communication mechanism. Instead, we assume a fairly basic event multicast service and do not rely on the lower-level communication infrastructure to provide sophisticated ordering and delivery options. We define a simple, high-level interface, which allows the underlying multicast implementation to be replaced transparently by the infrastructure.

The concrete service implementation we have developed in our prototype uses a centralized event multicast server, based on *Java*'s RMI mechanism, as the basic means of remote event delivery (Figure 4.8). Whenever the Coupler decides that an event must be transmitted, it passes it on to its local multicast client. The client makes a request for delivery by invoking the appropriate version of the server's `notify` method, with the event object to be delivered and the recipient(s), which can be a particular user or a whole group. Figure 4.8 shows an example, in which the coupler at site *A* must deliver an event to the rest of the group. In this case, as suggested by the single outgoing queue, the coupling policy is identical for all participants. Thus, the client invokes the `notifyOthers` method of the server, which places a copy of the event in the corresponding server delivery queues for *B* and *C*. If we had

separate coupling queues for *B* and *C* at the sender, then events would be unicast one at a time using the `notifyMember` server method.

To avoid deadlocks, as well as improve local response time, it is essential to make the notification mechanism asynchronous. For that purpose, the server maintains a separate queue and delivery thread (T_A, T_B, and T_C) for each connected client.

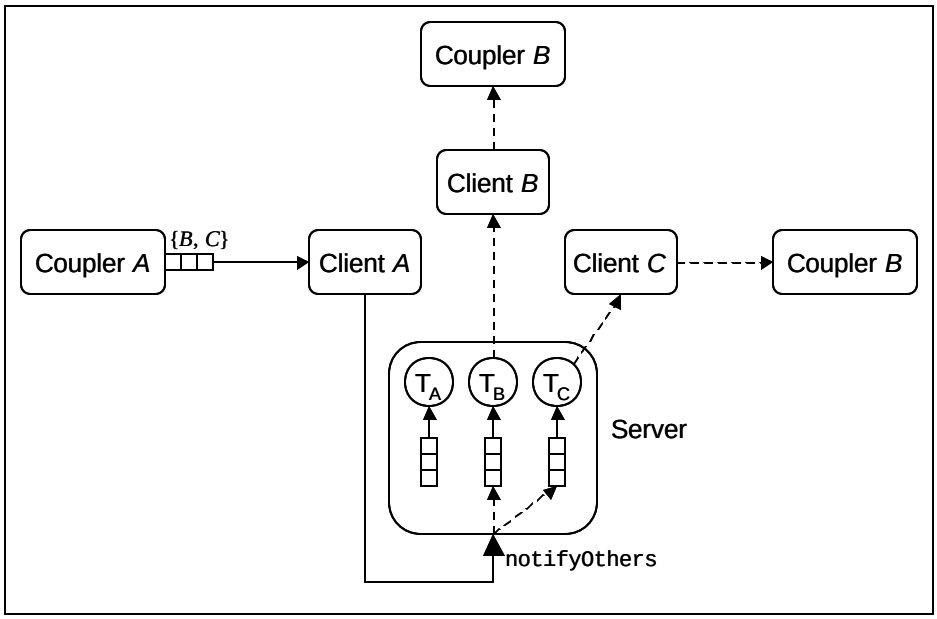


Figure 4.8 RMI-based Multicast Architecture

The sole function of a delivery thread is to contact its client and immediately deliver all events currently waiting in its queue. As soon as that is accomplished, the thread goes into a sleep state. Thus, whenever an event multicast request is received from a client, the `notify` method places a copy of the event into the queues of its intended recipients and immediately returns control to the caller. Before returning, the `notify` method wakes up all delivery threads with non-empty buffers so that they get a chance to run afterwards and deliver the contents of their buffers to their respective clients.

4.6 Session Management

A *collaborative session* refers to the virtual computer-mediated meeting of a group of users that allows them to work together on a shared object. *Session management* refers to the infrastructure function that allows users to create, join and leave a collaborative session. Thus, by definition, session management is an indispensable part of any multi-user

application. However, unlike other systems, such as *JCE*, we do not consider session management to be a separate concept from object sharing. Instead, like *GroupKit*, we have chosen to model it as a shared structure and apply the described sharing mechanisms to it.

Specifically, a session is a string name with an associated dynamic list of users maintained by a `SessionServer` object, which runs at a well-known location. `SessionClients` (representing different users) join/leave the session by contacting the server, which in turn expands/reduces the list of users and sends corresponding notifications to the rest of the users on the list. Such notifications can be used as triggers for collaborative functions such the transfer of the current state of a shared object to a newcomer.

4.7 Initialization

To complete our implementation description, we show the initialization steps that an application developer has to perform in order to use the infrastructure. Our discussion is based on an actual piece of code written for the GEF (Graph Editing Framework) sharing example, the details of which are described in the following chapter. For now, we will merely state that GEF is a non-trivial piece of software and its initialization code shown below is typical of the effort required to compose the single-user application with the sharing services of our infrastructure.

```
public class GraphDraw {
 public static void main( String argv[]) throws Exception {
   SystemBoot.init( argv);  // Command line processing
   SystemBoot.specsInit();  // Property specs loading
   SystemBoot.clientInit(); // Sharing services initialization

   // Original single-user startup code
   String frameTitle = "GraphDraw: " + System.getProperty("argv.u");
   JGraphFrame jgf = new JGraphFrame( frameTitle);
   jgf.setToolBar( new SamplePalette());
   jgf.setBounds( 0, 0, 400, 300);
   jgf.setVisible( true);

   // Root object registration
   DefaultGraphModel model = (DefaultGraphModel)jgf.getGraphModel();
   PropertyRegistrar.registerPersistent( model, "Graph");
   Layer figs = jgf.getGraph().getEditor().getLayerManager().getActiveLayer();
   PropertyRegistrar.registerPersistent( figs, "Figures");

   // Event adapter installation
   new GraphModelAdapter( model);
   new EditorModeAdapter( jgf.getGraph());
```

```
    // Object browser initialization
    CouplingObjectBrowser.addRootObject( model);
    CouplingObjectBrowser.addRootObject( layer);
    CouplingObjectBrowser.setVisible( true);

    // Collaboration menu attachment
    ColabJMenu.addColabMenu( jgf);
   }
}
```

The first three lines of the startup code—the `SystemBoot` invocations—provide general infrastructure initialization that does not depend on the particular application. The first two steps—command line argument processing and the loading of property specifications—are optional in that the application may elect to implement them differently. However, by using the set of infrastructure-defined command line parameters, such as the location of the property specifications, the application can pass onto the infrastructure the handling of such details that are largely irrelevant to the application's functions

Next, the general application initialization (which is independent of the sharing) is performed, followed by the registration of the root objects through the `registerPersistent` calls on the infrastructure. (Note that the creation the `model` and `figs` object happens as a result of instantiating the `JGraphFrame` earlier. Here, we merely obtain references to these already initialized objects. The `Layer` object refers to *GEF*'s own visual layers and is not related to our notion of application layers, which are given only as *XML* specification and never show up in the application code.) We have also developed two adapter objects that translate application-specific events into property and synchronization events and these adapters are installed next. Finally, the startup is completed by initializing the object browser and attaching a 'Collaboration' menu to the application's standard user interface.

4.8 Performance Considerations

Since our design relies on adding (multiple) levels of indirection at various points on the critical path of event processing, we need to evaluate the impact of this performance overhead on the end-user experience. Ideally, we would like to measure end-to-end delay experienced by the user and break it down into its components. In practical terms, however, this turns out to be a difficult task because there are multiple threads participating in the event processing and the scheduling decisions are beyond our control. Furthermore, our

analysis shows, in almost all cases, the added overhead cannot be adequately measured because it falls well below the granularity of the *Java* time routine.

Thus, we take a bottom-up approach, in which we measure the basic cost of the overhead added at each point of the event processing and produce a conservative quantitative estimate of the accumulated overhead.

4.8.1 Run-time Overhead

Our extensibility requirement has lead us to an implementation, which commonly replaces a direct method invocation with a table lookup, followed by a reflection-based method invocation (e.g., looking up and invoking a property handler). Therefore, measuring the cost of these operations is our first step in estimating our overhead. For that purpose, we used test programs that allowed us to obtain long-term averages over a large number of operations (10^6-10^8). All tests were performed on a system with a 1.6GHz Pentium 4 processor running *MS Windows XP* and version *1.4.0* of the *Java Runtime Environment* from *Sun Microsystems.*

4.8.1.1 Table Lookup

Throughout our implementation we have consistently used the standard `java.util.Hashtable` for storing table information. Thus, we are interested in the performance of the basic `get` and `put` operations. As it turns out, both of these are quite affordable. On average,

- A *get* takes between 1.2×10^{-3}ms and 1.4×10^{-3}ms as the size of the table varies between 100 and 500,000 entries;
- A *put* takes between 1.3×10^{-3}ms and 1.9×10^{-3}ms as the size of the table varies between 100 and 500,000 entries.

4.8.1.2 Reflection-based Method Invocation

To measure the cost of reflection-based invocation, we compare the time it takes to invoke an empty method with three arguments using direct invocation and reflection-based invocation (the exact number of arguments only marginally affects execution time).

On average,

- A direct method invocation takes approximately 1.1×10^{-5}ms;

- A reflection-based method invocation takes approximately 1.7×10^{-4}ms.

The above numbers, show that, in relative terms, a reflection-based invocation is quite costly compared to a direct one. However, in absolute terms, it is clear that reflection is quite affordable, even after multiplying the above numbers by a factor of 6 to account for the fact that the test program fits in the CPU cache entirely. We obtain the factor as the ratio of CPU cache speed (1.6GHz) and main memory speed (266MHz).

4.8.1.3 Policy Lookup

Given the hierarchical approach to specifying policy lookups, a coarse-grain specification (e.g., a single policy for the root object) can make the policy lookup expensive for large application structures. For example, suppose that there is a single policy specification given for the AWT `Component` class, that provides a default for UI objects. Now consider a policy lookup for a property of a `JCheckBox` of the standard *Swing* library (this could be triggered by a user clicking on the checkbox). Using the default structure-first traversal policy, it would take at least 10 lookups to find a policy: The structural hierarchy must be at least 3 levels deep (immediate container, root container, and bounding frame), whereas the `JCheckbox` class is 5 level below `Component` in the class hierarchy. In addition, a lookup for the affected property and the `JCheckBox` itself would also be performed. Hence, it is desirable to avoid paying such costs repeatedly.

Therefore, we use a policy cache, which retains the results of recent policy lookups. Furthermore, by taking into account that policies are typically specified at the object-level granularity (rather than property-level granularity), we can further improve the efficiency of the caching. For that purpose, whenever we lookup a property-level policy and the parent object does not have a specified policy, we cache the result both at the property and the object level. This effectively pre-fetches the policy for other properties of the same object. For example, if the user modifies the *position* of a shape object in a drawing, followed by a change of the *size*, the policy lookup for the *size* property would take a maximum of two steps.

4.8.1.4 Remote Communication

The overhead of communicating property events across the network consists of two major parts: object serialization and transmission. The former represents the cost of

converting a *Java* object into a suitable byte representation that can be transmitted over a TCP/IP socket connection. The latter is the actual time required for the transmission itself. Since the transmission cost is dominated by network characteristics and will have to be paid by any infrastructure, we focus our attention on the serialization cost as it is a function of the specific approach we have taken.

We performed a series of tests to quantify the cost of serializing a `PropertyOperation` object, which carries the property updates across the network. The time it takes to serialize/deserialize such an object, carrying three operation arguments was approximately `0.03ms` in our experimental setup. Given that our centralized multicast service implementation requires an extra serialization/deserialization cycle at the server, we estimate that the overall serialization overhead we incur on the critical path of a property event is about `0.12ms`. Thus, serialization has a marginal impact on user experienced end-to-end delay.

4.8.2 Pattern Analysis

Another component of the processing overhead is the initial pattern analysis of shared objects, which based in the programmer-provided specifications, discovers the properties of an object. To estimate the cost of this analysis, we measured the amount of time it takes to perform it under different circumstances. The complexity of the current (un-optimized) version of the analysis algorithm is approximately proportional to the product of the number of methods of the analyzed object and the number of property specifications. Therefore, we performed a number of tests on actual classes from the *JDK* with the actual property definitions we have used in various applications.

In the table below we present, what we consider, extreme cases to illustrate the range of observed executions times. We have chosen two commonly used classes—one with few methods and properties (`java.util.Vector`) and one with numerous methods and properties (`javax.swing.JFrame`)—and measured the actual time it took to perform the analysis using `1` and `12` property definitions (the maximum number we have used in our application implementations).

Class	Number of Methods	Number of Specifications	Number of Properties	Execution Time (ms)
`java.util.Vector`	50	1	1	60
`java.util.Vector`	50	12	1	100
`javax.swing.JFrame`	286	1	27	180
`javax.swing.JFrame`	286	12	33	320

Table 4.1 Pattern Analysis Execution Times

As the numbers in Table 4.1 suggest, pattern analysis is expensive but practical given the fact that it is a one-time effort for a given class. However, if a relatively large number of classes must be analyzed at once (e.g., at startup), this can cause a user perceived 'pause' in the execution. Therefore, once the analysis is performed, we cache the results persistently to make them readily available for the next execution. This technique can be taken a step further by performing the property analysis entirely off-line as a post-compilation step, thereby minimizing the computational overhead associated with pattern analysis.

4.8.3 Summary

In summary, the measurements of the intrinsic steady-state costs (those incurred after the initial pattern analysis) incurred by the implementation show that the impact on user experience is negligible, even by our most conservative estimates. For example, if the infrastructure performs 100 put operations and 100 lookups on the critical event path—a *highly* unlikely case—at the cost of $2x10^{-3}$ms and $1.5x10^{-3}$ms, respectively, the overall effect on end-to-end delay would still be no more than `0.3ms`. Combined with an estimated serialization cost of about `0.12ms`, we obtain as a conservative estimate of the infrastructure-incurred computational overhead of less than `0.5ms`. Thus, on a fast local area network with round-trip times in the order of `0.25ms` (under good conditions) we can reasonably expect the estimated end-to-end delay to be below `1ms`. Clearly, on a wide area network, end-to-end delay will be dominated by the network delay, with typical round-trip times between 20ms (for the local metro area) and 100ms (coast-to-coast) [5].

We also showed that pattern analysis is an affordable operation that can also be performed off-line to further minimize its impact on performance. Finally, we should note

[5] Measurements taken with `traceroute`

that in our qualitative experience, the largest impact on user-perceived delay (on a fast local network) is the refreshing of the user interface performed by the *Java Virtual Machine* in response to user actions. Unfortunately, we were unable to obtain a direct measurement because of the asynchronous nature of such updates.

5. EXPERIENCE

In this chapter, we present our experience in building collaborative applications using our prototype infrastructure implementation, as well as the use of pattern-based approaches to problems not related to collaboration. In the collaboration side, we start with the outline example and work through the details of implementing various sharing scenarios. The outline application was chosen as an example of a structured text editor that is comparable to the editors supported by our benchmark system for sharing flexibility—*Suite*. Next, we describe our shared user interface toolkit implementation, which is an example of UI sharing for applications based on *Java*'s *Swing* library and is directly comparable to another benchmark system—*JCE*. Next, we use describe our experience with the *GraphDraw* and *Shapes* applications. The two are structured graphical editors that have been developed by third parties and we use them demonstrate the effort in converting an existing third-party single-user application into a collaborative one. In all cases, our primary interest is to measure the extent and complexity of the development effort involved, as well as the respective flexibility benefits.

Next, we use three examples of generic non-collaboration infrastructure services that we have developed based on our abstraction and architectural models. These include *XML*-based serialization of *Java* objects, *UPnP* system interface generation, and *UPnP* user interface generation. In all cases, our pattern-based approach yields significant automation gains and provides more general solutions than currently possible.

5.1 Outline Application

Our discussion of the *Outline* application follows the implementation of different sharing scenarios. We begin with the most basic sharing scenario, which requires the least amount of development effort and has the least amount of flexibility, and we gradually increase the sophistication of the application's sharing capabilities and present the respective development effort.

Our starting point is a relatively straightforward single-user implementation of the *Outline* application. It is based on two objects—*model* and *view*—with the former being a concrete implementation of the skeleton code shown throughout our presentation and the latter using a `JTree` component from the standard *Java AWT/Swing* user-interface toolkit to render and edit the model. In other words, the `Outline` object represents the model layer of the application, whereas the `JTree` represents the view layer, as shown below:

Model

```
Public class Outline {
  String getTitle();
  void    setTitle(String title);

  void    insertSection(int i, Section s);
  void    removeSection(int i);
  Section getSection(int i);
  void    setSection(int i, Section s);
  int     getSectionCount();
}
```

View

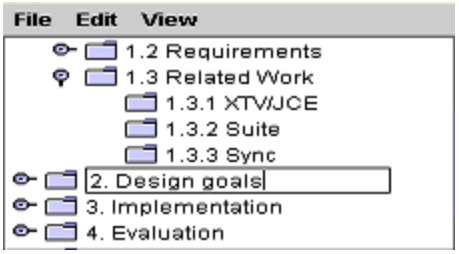


Figure 5.1 Model-View decomposition of the Outline application

The `Outline` object also implements a number of methods from the standard `MutableTreeNode` interface, which facilitate its interaction with the `JTree` object. Since these methods are routine and irrelevant to our discussion, we have opted to leave them out. Similarly, we skip many other details, such as the precise mechanisms used by the application to handle insertions/deletions of sections, that have no bearing on our sharing problem.

5.1.1 Asynchronous Sharing

Our first group of scenarios represents the model-level sharing of the `Outline` object without assuming update notification. The rationale here is that the baseline (single-user) implementation does not require the `Outline` object to issue update notifications so we should explore the possibilities under this condition first.

5.1.1.1 Scenario 1: Read/Write Peer-to-Peer Sharing

The lack of a notification mechanism implies that we are restricted to asynchronous sharing, in which the infrastructure must poll the shared object to obtain updates. In its simplest form, such polling is reduced to obtaining the current state of the replica. In our

model, this is specified through the use of a sharing policy that has a *read* value for its acquisition parameter. For example, we can specify a `<Read, Increment, Raw, Replay>` policy for the `Outline` object, which indicates that updates are acquired by *read*-ing the object, then are transmitted with no constraints with respect to the communication operation (`Increment`) or level of correctness (`Raw`), and the values are directly installed on peer replicas (`Replay`). (Throughout this chapter, when we give policies, we always consider the *effective* policies resulting from the combination of the outgoing policy of the sender and the incoming policy of the receiver.)

As already discussed in Section 4.3, the triggering mechanism for obtaining and communicating the update mechanism can be an explicit *Transmit/Commit* operation, timer expiration, or an application trigger (Figure 4.2). In the case of explicit initiation, one option for the user is to select the 'Transmit All'/'Commit All' item from the 'Collaboration' menu, which generates a synchronization event for each of the objects registered as root with the infrastructure; in this case the `Outline`. As a result, the coupler consecutively reads the current values of each of the object's properties and transmits them as (*over*)*write* property operations. The operations are tagged according to the transmit parameter of the triggering synchronization event—*Transmit* or *Commit*; the transmit parameter is set to *Scheduled* whenever the event is triggered by a timer.

Upon delivery at the remote site, the coupler looks up the respective *write* handlers for each property and uses them to overwrite the current values. In effect, this scheme is very similar to the classic file-based sharing and requires that users take turns at editing the outline to avoid overriding each other's changes. However, since the infrastructure is aware of the logical structure of the outline, we can do better than file-based sharing by transmitting parts of the shared structure. For that purpose, users can open the object browser, select the object they want to transmit/commit, and issue the respective command, as shown on Figure 5.2.

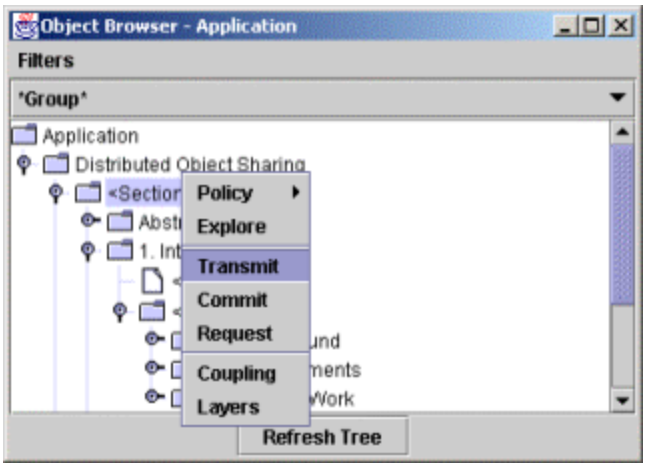


Figure 5.2 Selective Transmit/Commit

Thus, if user *A* works on section 1, while user *B* works on section 2, the two can exchange updates without overriding each other's changes by selectively sending the values of section 1 and 2, respectively.

To conclude our discussion on this scenario, let us summarize the development steps involved in achieving this type of sharing:

- Modify the `Outline` class to either implement the `Identifiable` interface, or subclass `ReplicatedObject`.
- Define two *XML* property specifications for *simple* and *sequence* properties, such as the ones discussed in the previous chapter. Since the infrastructure fully supports *simple*, *sequence*, and *table* properties, it provides default such specifications that can either be reused directly, or with minor revisions.
- Implement four property handlers: a *read* and a *write* handler for each of the two property types (*simple* and *sequence*). The infrastructure provides default implementations and those can be reused directly, or with minimal revisions, such as the reordering of arguments. The implementations themselves are rather straightforward and require between 10 and 20 lines of code each.
- Modify the startup code to include initialization of sharing functions. For this application, this translates into 4-5 additional lines of code, as shown below (in bold):

```
public static void main( String[] argv) throws Exception {
  SystemBoot.initAll( argv);
  // Single-user initialization
  Outline outline = initOutline();
  JFrame outlineView = initOutlineView( outline);
  outlineView.setVisible( true);
  // ------------------------
  ColabJMenu.addColabMenu( outlineView);
  PropertyRegistrar.registerPersistent( outline, "Outline");
  CouplingObjectBrowser.addRootObject( outline);
  CouplingObjectBrowser.setVisible( true); // optional
}
```

5.1.1.2 Scenario 2: Diff-Based Peer-to-Peer Sharing

Although we can leverage the knowledge of the outline's logical structure to implement fine-grain sharing, the read/write model may be too inflexible to accommodate user needs, even if they are satisfied with the general idea of asynchronous sharing. For example, consider the insertion of a new top-level section into the outline—it requires that the state of the whole outline be transferred for the insertion to be shared. Moreover, it also implies that receiving users must stop working while the insertion is performed and communicated to avoid their changes being overwritten. More generally, users must carefully coordinate their actions and must keep track of the changes they make to avoid rolling back other users' actions.

To avoid such problems, a developer may use the infrastructure-provided object diff-ing service. From a user's point of view, the only difference is the policy specification of the acquisition parameter—instead of *Read* it is set to *Diff*. From a developer's point of view, the effort is very similar to that of the previous case. The only difference is that instead of *write* handlers, the developer must provide *diff* handlers. Arguably, these require a somewhat greater effort—the default implementations supplied by the infrastructure are between 50 and 100 lines of code. However, since diff-ing discovers user updates at a finer granularity, it also allows the infrastructure to make better decision in interleaving them so that user intentions are preserved. Moreover, we expect the default diff handlers to be used in most applications.

Recall that, to achieve this finer granularity, the diff-ing is performed in a recursive manner. Hence, it would be necessary, in addition to the `Outline`, to analyze also the `Section` object. Fortunately, depending on the concrete implementation chosen, this requires little or no extra effort in our infrastructure. To illustrate the point, let us consider two options for the implementation of `Section` and their respective costs to the developer

The first one is to establish an *Is-A* class inheritance relationship between the `Section` and the `Outline`. Indeed, since they have similar logical structure and are related, it is reasonable to move the implementation of the basic functionality (the seven methods shown on Figure 3.1) into the `Section` and for the `Outline` to inherit it. In other words, the design establishes that the `Outline` *Is-A* `Section`. Like other infrastructures (e.g., *Sync*) ours requires no additional effort to share a subclass, if the superclass is already shared. Thus, if everything is in place for the `Section`, the `Outline` will automatically be shared.

However, a second line of reasoning is to conclude that, while the `Outline` *Has-A* number of `Sections`, it is not a special case of a `Section` because, eventually, its logical structure may differ significantly in that it may have to maintain a lot of other information, such a list of authors, a set of keywords, etc. Thus, we may end up with two unrelated (in terms of inheritance) classes, which partially share logical structure but may have deifferent implementations. For example, the `Section` class may look as follows:

```
public class Section extends ReplicatedObject {
  public String getHeading();
  public void   setHeading( String heading);
  public Section[] getSubsections();
  public void      insertSubsection( int i, Section s);
  public void      removeSubsection( int i);
}
```

Thus, we will need a second version for the sequence property specification, as well as a (trivial) new property *read* handler. However, we would be able to directly reuse the most expensive part of the implementation—the *diff* handler—by appropriately structuring its implementation. To illustrate, consider the following template for the *sequence diff* handler:

```
SequenceDiffer(Object obj1, Object obj2, Property property) {
  PropertyReader reader = (PropertyReader)property.getHandler(“read”);
  Object[] value1 = reader.getValue(obj1,null);
  Object[] value2 = reader.getValue(obj2, null);
  // Compute diff of value1 and value2 …
  …
}
```

By using the *read* handler to abstract away the details of the read operation, we were able to fully reuse the *differ* across multiple *sequence* property implementations, thereby minimizing developer’s effort.

5.1.1.3 Scenario 3: Centralized Commit-Based Sharing

Another extension to our first scenario is to supplement the peer object replicas with a centralized master copy, which keeps only the committed state of the shared object (Figure 5.3). This approach is analogous to the *Suite*'s sharing model, although the implementation is achieved differently. Conceptually, the master copy, which corresponds to *Suite*'s active object, is just another replica of the shared object. Its main characteristic is that it accepts only those incoming changes that are marked as *committed*. Specifically, it has the following *fixed* sharing policy: `<Read, Increment, Commit, Replay>`.

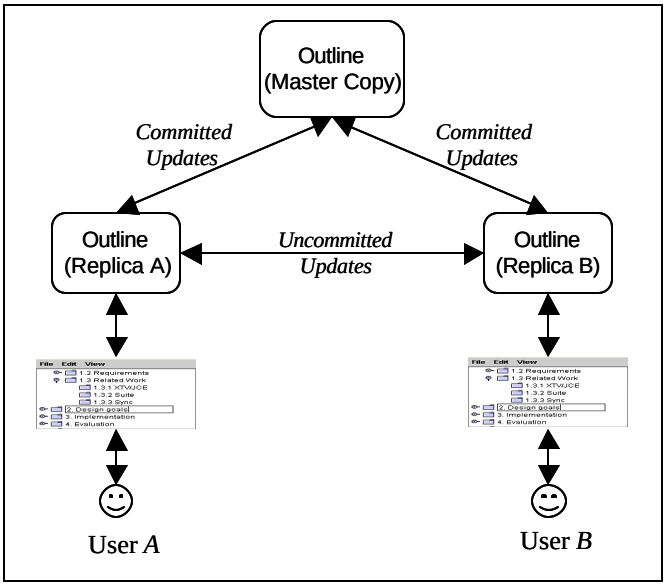


Figure 5.3 Centralized Commit-Based Sharing

Since the master copy does not interact with the user directly, it runs a modified version of the application code that does not include the user interface part. Thus, the master copy also never *initiates* updates, which could become a problem if a user working on his local (non-committed) version decides to revert to the committed one. To support this case we give the user the option to request the current state of an object by selecting it on the object browser and issuing a *request* command. The request, which resembles *Suite* install command, may be directed at any of the participating users or the master copy. From an implementation point of view, the request is simply a synchronization event that is triggered by the actions of a remote user rather than the local one. When the request is received by the master copy, it uses the current acquisition method—*read* in this case—to obtain the current state of the object and send it back to the requesting user. If the acquisition were set to *Log*,

the corresponding outgoing queue of events would be flushed, which is the semantics implemented by *Suite*'s *install*.

From the discussion so far, it should be clear that the decision to use a peer-to-peer or a centralized architecture for the shared object is independent of the mechanisms used to obtain the updates. Therefore, the described master copy approach can be used for asynchronous sharing both with *read*-based and *diff*-based policies.

To summarize the developer's effort—it is the same as in the corresponding peer-to-peer case (Scenarios 1 and 2) with the additional need to set up and run a master copy. In the *Outline* example, the master copy version is implemented by simply omitting the UI initialization in the application startup code, which reduces it to the following:

```
public static void main( String[] argv) throws Exception {
   SystemBoot.initAll( argv);
   // Single-user initialization
   Outline outline = initOutline();
   // ------------------------
   PropertyRegistrar.registerPersistent( outline, "Outline");
}
```

5.1.2 Flexible Event-Based Sharing

So far we discussed scenarios in which the multi-user application code is practically the same as the single-user version except for the implementation of the `Identifiable` interface by shared objects and the inclusion of collaboration services in the startup code. Let us now consider the case where the application programmer provides specific collaboration support in the form of a notification mechanism, which may, in fact, be provided even in a non-collaborative application to componentize it.

5.1.2.1 Scenario 4: Peer-to-Peer Sharing

The basic precondition for implementing event-based sharing is that the application should issue an appropriate property event whenever a change to the state of a replica is introduced. In our application, this translates into issuing a `PropertyOperation` event after the invocation of each of the four methods that modify the `Outline` object—`setTitle`, `insertSection`, `removeSection`, and `setSection`. In practical terms, this can be implemented in five lines of code as follows. The first step is, as part of the object initialization, to obtain from the registry a reference to the coupler, where all events are to be delivered:

```
public class Outline extends ReplicatedObject {
  ColabEventListener coupler = (ColabEventListener)
        CentralRegistry.lookup("service.CentralEventDispatcher");
  ...
}
```

Next, for each of the four modifier methods, add a statement issuing a notification before completing the method invocation. For instance:

```
public void setTitle(String title) {
  …
  coupler.dispatch(new PropertyOperation(this.getGID(), "title",
                                         "setter",new Object[] {title}));
}
```

Once the coupler receives the notification, the rest of the sharing is handled automatically based on the current policies in place. Thus, the basic investment into a notification mechanism on part of the developer is fairly modest. Yet it provides additional flexibility to the user.

First, it allows asynchronous sharing, similar to Scenarios 1-3, to be implemented through event logging/replay: by default, all property events are `Increment` and `Raw`; if the effective policy specifies that only `Transmit/Commit` events should be communicated, then exchange takes place only upon the explicit user command. Similarly, `Scheduled` events are transmitted at user-defined times.

Second, it allows for synchronous semantic sharing of the outline, which can be achieved by setting a sharing policy that allows all incremental updates to pass through without placing restrictions on their correctness.

To take full advantage of the sharing options presented by our model, the application must also provide additional information about the level of correctness of the updates and the communication operation executed. Generally, the model-level object (i.e., the `Outline`) must be aware of the level of correctness of each change, as it is its job to ensure it. Hence, it only needs to pass along this information by tagging the updates as `Parsed`, or `Validated`. In terms of implementation, this corresponds to one more line of code for each modifier method:

```
...
PropertyOperation po = new PropertyOperation( outline,"sections",
                                              "insert", newSection);
po.setCorrectness(2); // extra line: set correctness to Validated
coupler.dispatch( po);
...
```

Indicating a `Complete` editing operation (the only value of the transmission parameter not covered by asynchronous sharing) is slightly more complicated, as it involves the user interface. One of the inexpensive ways to realize this in our example is to replace the default editor for the `JTree` nodes—a text field editor—with a custom one that, in addition to performing the editing operations, also issues synchronization events whenever the editing is completed. By subclassing the standard *Swing* `DefaultTreeCellEditor`, the actual implementation takes about 15 lines of additional code.

In summary, depending on the desired level of support, we need 20-35 lines of application code to incorporate event-based sharing, in addition to the two property specifications and two *read* handlers, which are identical to the previous cases.

5.1.2.2 Scenario 5: Centralized Sharing

This scenario extends the previous one in the same way as Scenario 3 extends Scenario 2 in the asynchronous case, by adding a master copy, which holds the committed state of the shared object. The cost is *identical* to the one discussed in Section 5.1.1.3 and includes the creation and setup of a version of the code that has no user interface (the exact same one used in Scenario 3).

5.1.2.3 Scenario 6: Model/View Sharing

So far we have only considered the sharing of the semantic `Outline` object and not its user interface presentation. While semantic sharing is likely to cover a significant fraction of actual usage scenarios, it is not always sufficient. For example, in our implementation, the `setTitle/setHeading` methods of the `Outline/Section` objects are not invoked until the user has finished editing the corresponding field. However, it may be desirable for users see and discuss incremental changes, such as inserting a character, as they happen. Furthermore, observing changes also provides implicit awareness of other user's actions, even if the individual changes are not that important. For example, user *A* may observe that some other user is editing section 2 and decide to work on a different section.

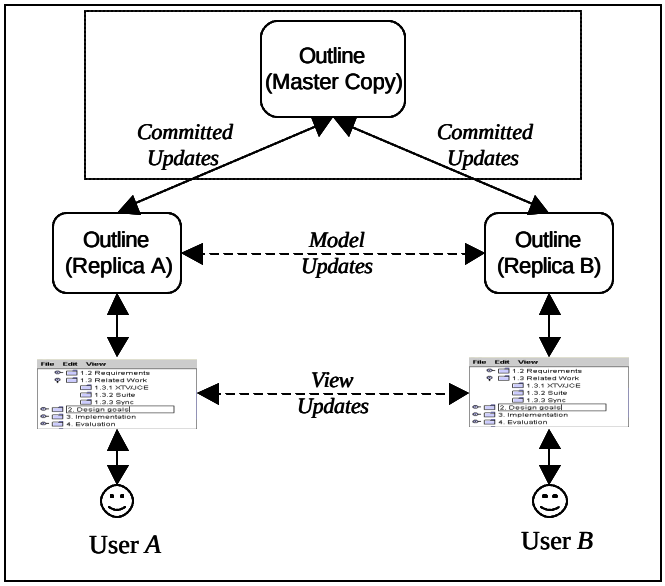


Figure 5.4 (Centralized) Model/View Sharing

Let us now consider the scenario depicted on Figure 5.4, which extends previous ones by incorporating the sharing of the view layer. The dotted rectangle in the picture represents the fact that we consider both the peer-to-peer and centralized versions of this sharing scenario. The effort to create and manage the master copy is identical to the previous centralized scenarios. However, there are some minor differences in the event handling in the two cases, which we will point out in due course.

Recall that we use *XML* definitions to describe the application layers as well as their inter-dependencies. In our simple example, the application consists of three classes—`Outline`, `Section`, and `JTree`—with the first two comprising the *model* layer and the last one the *view*. Since the *model* (i.e., the `JTree`) is the editing interface through which all changes to the *model* are introduced, we consider the *model* to be dependent on the *view*. Following the *XML* definitions described in Chapter 4, we specify this application layering as follows:

```
<?xml version="1.0" encoding="UTF-8" ?>
<object class = "outline.Outline">
  <layer name="model">
    <ALL />
  </layer>
</object>
```

```
<?xml version="1.0" encoding="UTF-8" ?>
<object class = "outline.Section">
  <layer name="model">
    <ALL />
  </layer>
</object>
```

```
<?xml version="1.0" encoding="UTF-8" ?>
<object class = "javax.swing.JTree">
  <layer name="view">
    <ALL />
  </layer>
</object>
```

```
<?xml version="1.0" encoding="UTF-8" ?>
<layer_dependencies>
  <view>
    <model/>
  </view>
</layer_dependencies>
```

Recall that the basic problem we address though layering description is to avoid multiple notification of causally related events. Such events occur as a result of the transformation of user actions from one layer of abstraction to the next. Thus, given the layer dependencies and a specified shared layer, the infrastructure must ensure that events from the specified shared layer get through, while filtering out events received that are dependents of or editors of the given layer are filtered out. Translated into our example, this means that whenever model sharing is chosen, no view updates should be communicated. Conversely, if view sharing is chosen, then no model updates should go through.

The latter rule, however, has an important exception when the application has a master copy. In this case, model updates (identified by the fact they belong to a layer with no dependents[6]) are specifically communicated to the master copy but not other replicas (by conventions, the master copy is one with the special name of `"*MasterCopy*"`). More generally, the master copy must always receive committed model updates regardless of the sharing policy. The reasoning here is fairly straightforward—because the master copy belongs to one layer, there is no chance of multiple notifications. Furthermore if model updates are not delivered even when the model layer of not shared, the master copy will be of no use.

---

[6] This may lead to the communication of some non-model events, which are ignored if they refer to an object that does not exist at the master (e.g., window). A planned extension of the layer specification mechanism will solve this by specifically tagging the layers that are present at the master copy.

The user view of the shared layer specification is very simple—the user pulls up the "Collaboration" menu and selects the "Shared Layer" item, which brings up the tree representation of the layer dependencies shown on Figure 5.5 (prefix "[Increment]" marks a currently shared layer). A right-click on the layer name pops up the list of available policies and, upon selection, the sharing for dependent layers is automatically reset.

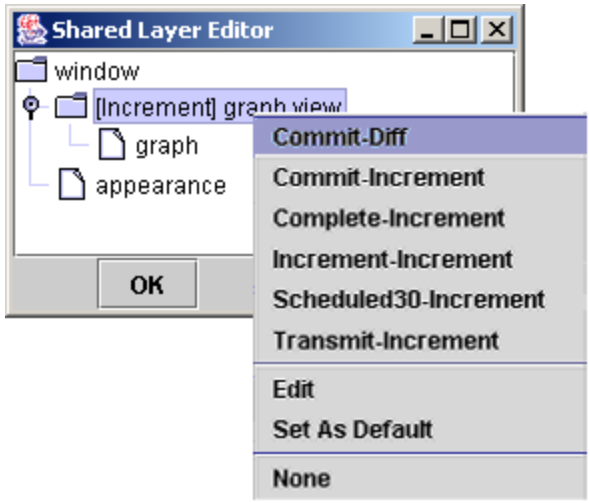


Figure 5.5 Layer Sharing Specification User Interface

The next step is to connect the view and the coupler so that the view can deliver notifications. As before, this can be achieved by looking up the coupler, which is registered under a well-known name from the registry. However, this would require the programmer to implement the translation of `JTree` events into property events. To automate this process, the infrastructure provides an `AWTEventAdapter` object, which is a static object, and automatically translates all *AWT/Swing* events it receives into property operations and delivers them to the coupler. Thus, the connection is reduced to making the `AWTEventAdapter` a listener of all relevant `JTree` events:

```
AWTEventAdapter.translate(jtree);
```

The AWT event adapter is initialized at startup time and installs itself as a listener of the system event queue, through which all events pass. It also maintains a table of objects for which the translation must be performed. Whenever an event is received, the source is looked up in the table and, if present, the event is translated. Thus, the above `translate` invocation simply adds the jtree object to the table. The adapter also has methods for translating all events and for removing specific, or all, objects from the table.

Finally, we need to register the `JTree` object with the registry to establish its global name mapping. This achieved by adding the following line to the startup code:

```
PropertyRegistrar.registerPersistent(outlineView, "View");
```

To summarize, the effort to add view sharing option to the model sharing of previous scenarios consists primarily of providing appropriate layer definitions and registering the view objects. The translation of the standard UI events to property events is largely automated by the provided adapter.

5.2 User Interface Toolkit Sharing

Conceptually, our UI toolkit sharing has the same goals and has similar mechanisms to that of our benchmark system—*JCE*—however the actual implementation is somewhat simpler[7]. Most notably, unlike *JCE*, we do not replace the standard implementations of the *AWT/Swing* components so that we can intercept their event stream. Instead, we use a specialized version of the standard `GlassPane`[8] container, called `SharedGlassPane`, which we place on top of every shared frame, as illustrated by Figure 5.6.

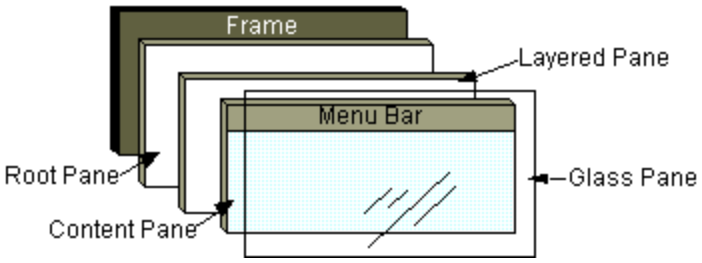


Figure 5.6 *Java*'s GlassPane container

The main purpose of this graphically transparent object is to gain complete control over the capture and replay of all user actions. Whenever the `SharedGlassPane`'s *transparent* property is set to `true`, the UI toolkit delivers all mouse/keyboard events to it, and allows the infrastructure to decide how the event is processed. In particular, the events initiated by the active user are allowed to proceed as usual and are also replicated at all other sites. At the same time, local events of inactive users are blocked, which guarantees the exclusivity of

[7] Some of the *Java* features we use were not available at the time when *JCE* was first developed.

[8] http://java.sun.com/docs/books/tutorial/uiswing/components/rootpane.html

user access required to ensure the correctness of UI sharing. Thus, we effectively share the queue of UI events generated by the active user's actions.

A secondary purpose of the `SharedGlassPane` is to support the use of *telepointers*, such as the ones provided by *GroupKit*. A telepointer is simply a local presentation of a remote user's pointer/cursor (usually with a different shape or color to help distinguish it). The only notable difference in our implementation is that it is provided (and managed) as part of a shared UI system, whereas in *GroupKit* it is provided as a widget that must be explicitly managed by the application.

In our implementation, the only change required to the standard UI library is to make all UI components `Identifiable` so that peer objects can be identified and events are delivered properly. Since all such components are subclasses of `java.awt.Component`, it is the only one we have modified and replaced in the standard library.

Like *JCE*, the naming of UI components is handled automatically by the infrastructure in cases where window sharing is used throughout an entire collaborative session. However, if dynamic multi-layer sharing is to be supported, the application programmer may need to provide some minimal support. In particular, if the application creates more than one application window, the developer must explicitly provide names for all root windows. Otherwise, the infrastructure may be unable to conclusively match application window replicas after a period of non-window sharing.

In summary, given the scenario of continuous window sharing, our infrastructure provides the same features and development effort as *JCE*. In addition, we support a transition scheme that allows the application to switch from non-window to window sharing. In this case, the programmer may need to provide minimal additional support in the form of naming. We should also note that such a transition may not always be possible—if one user has two application windows opened while another only one, it is not possible to reconcile the two versions regardless of who initiates the exchange. (Simply recreating the missing window would not solve the problem because the UI components would not be connected to the respective application objects being rendered and we would end up with a "mockup" window.)

5.3 GraphDraw Application

5.3.1 Overview

The *GraphDraw* application is a sample application provided as part of the *GEF* (Graph Editing Framework) developed at the University of California at Irvine by J.Robbins. The basic goal of *GEF* is to provide a UI toolkit for the development of various applications requiring graph editing, such as circuit design, or a *Petri* net editor. All of our experiments were performed with version 0.6 of the implementation. A more recent version of *GEF*[9] serves as the UI platform for the open source *ArgoUML* editor[10].

The *GraphDraw* application (Figure 5.1), consisting of 9 *Java* classes, is fairly simple in that it defines two types of graph nodes and two types of graph edges and registers them with the framework, which handles everything else. Thus, the essential task we performed and measured was the transformation of *GEF* from a single-user toolkit into a multi-user one. The main reason we chose *GEF* was that it made extensive and consistent use of programming patterns and, therefore, made a good test case for our assertion that we can provide a low-cost high-flexibility sharing solution based on patterns for existing code.

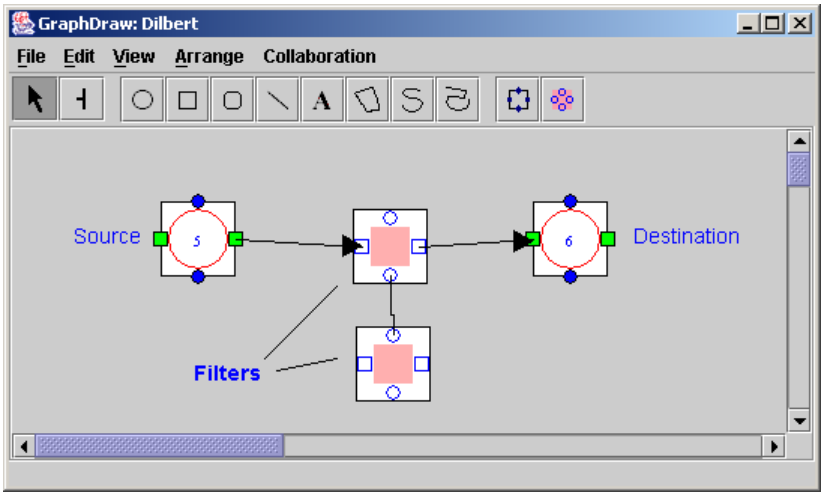


Figure 5.7 *GraphDraw* Application

Overall, *GEF* consist of `171` *Java* classes totaling over `26,000` lines of code. A *GEF* graph diagram consists three basic layers—a *graph* layer, a *graph view* layer, and a *figures*

[9] http://gef.tigris.org/

[10] http://argouml.tigris.org/

layer. The graph layer represents the abstract graph (nodes and edge) being edited. Applications can add specific semantics to the nodes and edges (e.g., implement a *Petri* net) by reacting notifications whenever new nodes/edge are created, connected, disconnected, and destroyed. The *graph view* provides the specific graphic representation through which users can manipulate the nodes and edges of graphs. The *figures* layer consists of a number of standard shapes, such as ovals, rectangles, and text boxes that can be used to annotate the graph. In general, the graph and the figures layers are separate and may exist independently of each other—a graph may not have annotations and an annotations-only diagram is simply a drawing like the ones produced by any standard drawing editor. In addition, we define a *window* and an *appearance* layer much like we did for the *Outline* application. The *window* layer encompasses all objects within the application window, whereas the *appearance* layer consists of the window components/attributes that do not affect the editing of the graph—window size/position, scrollbar position, etc. The resulting decomposition is shown on Figure 5.8.

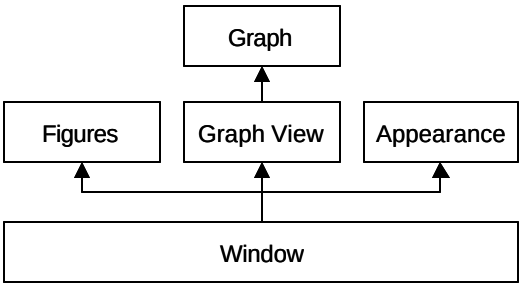


Figure 5.8 Layer Dependencies for *GraphDraw* Application

Although we have implement the peer-to-peer versions of all of the scenarios mention above, to avoid repetition, we focus the following discussion on two scenarios that we have not discussed so far—dynamic multi-layer sharing and window sharing.

5.3.2 Dynamic Multi-layer Sharing

Multi-layer sharing is the natural generalization of the model/view sharing discussed in the previous section. In the model/view case the sharing of one layer automatically precludes the sharing of the other. In multi-layer sharing, any combination of layers that are not dependent on each other may be shared at the same time. For example, any combination of shared *graph*, *figures*, and *appearance* layers is admissible and the judgment call about the usefulness any particular combination is left to the users (or applications). Consequently, it is

also crucial to allow users to explore sharing modes and dynamically switch on and off the sharing of each individual layer. Let us briefly consider some of the possibilities presented by different combinations:

1. Share *graph* layer only (Figure 5.9). In this mode, the topology of the graph would be shared but its visual appearance and annotations may differ. Note that, as a side effect of replicating the insertion of a graph node/edge into the graph, a corresponding view object will also be inserted. Initially, the insert position of the view object would be the same as the original; however, subsequent updates would not be reflected

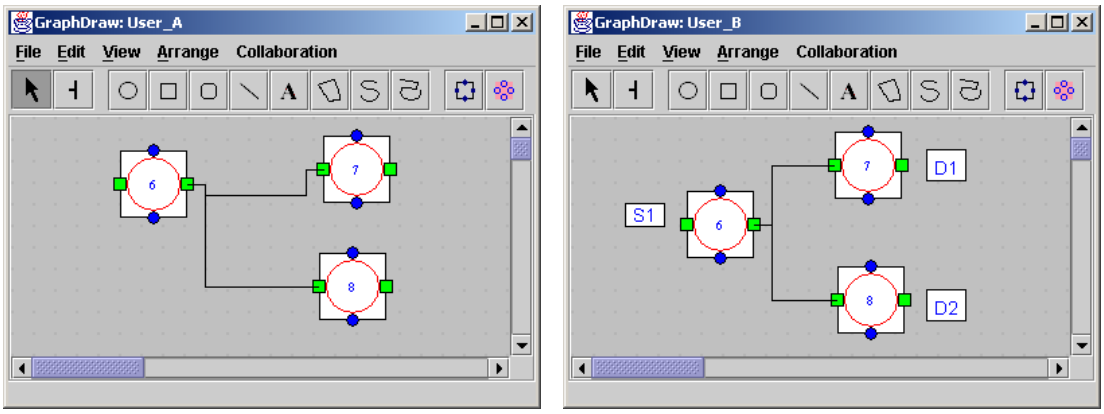


Figure 5.9 *GraphDraw* application: *graph* layer sharing

2. Share *graph view* layer only (Figure 5.10). This leads to the sharing of both the topology and the graphic presentation of the graph. Annotations, however, may be different. This is useful when different users work on different aspect of the graph and their annotations of the graph are largely private.

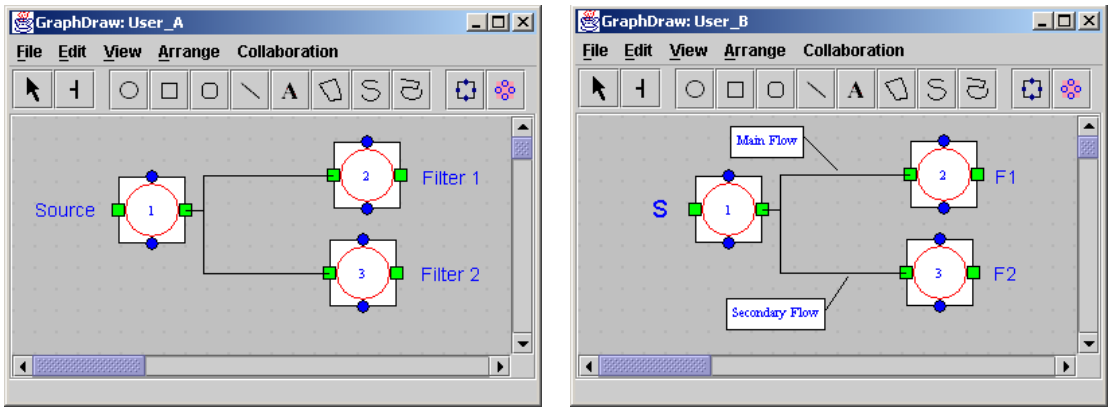


Figure 5.10 *GraphDraw* application: *graph view* layer sharing

3. Share *graph view* and *figures* layers (Figure 5.11). Essentially, the graphic representation of the graph with all its annotations will be shared in its entirety, however, users retain the freedom to navigate autonomously and edit the graph concurrently.

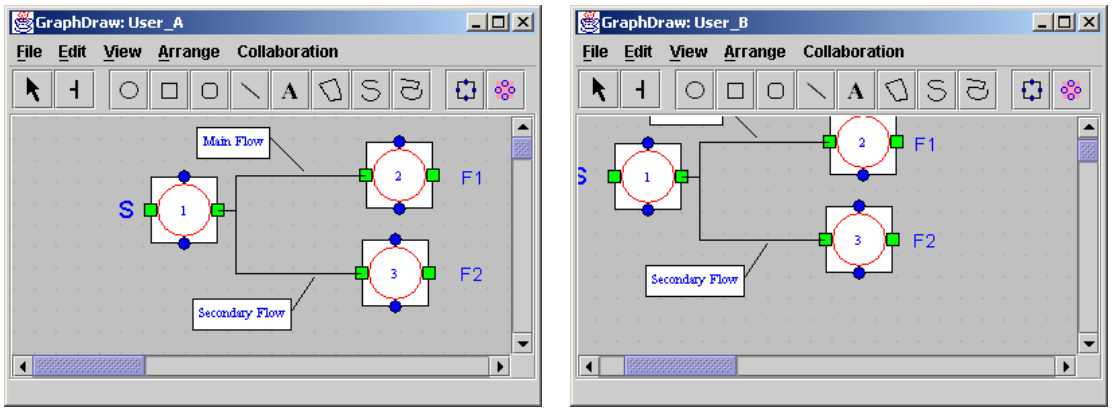


Figure 5.11 *GraphDraw* application: *graph view* and *figures* layers sharing

4. Share the *window* layer (Figure 5.12). This is a form of the UI-based sharing we discussed in Section 2.1—all users have identical views of the application and only one user at a time can be active. The details of this implementation are given in the next section.

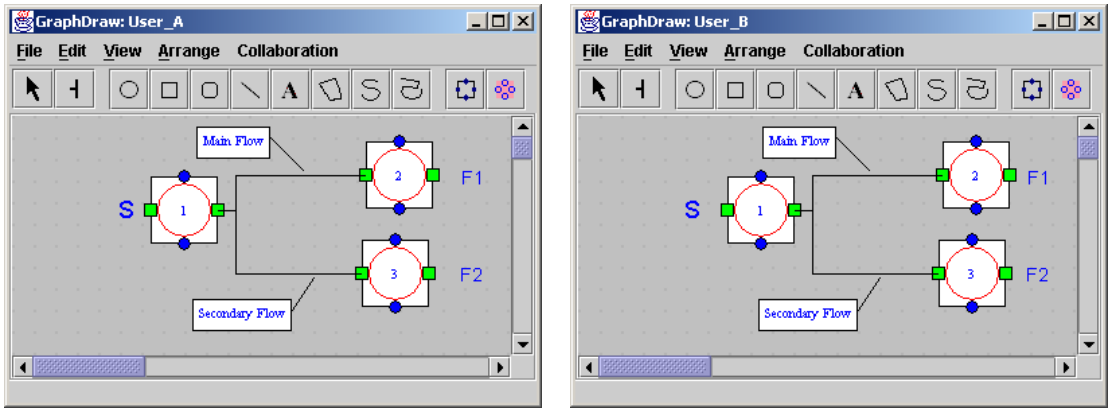


Figure 5.12 *GraphDraw* application: *window* layer sharing

Experience shows that the above sharing modes are the most likely to be used in actual collaboration and that the ability to switch from one mode to another is highly desirable [10, 11]. Therefore, let us now consider how the actual transition would be achieved. From a user's point view, the effort consists of clicking on the respective layer and selecting a sharing policy.

From the point of view of the infrastructure, the transition from a higher-numbered (in the above list) sharing mode to a lower one is trivial—the infrastructure simply stops sharing the updates of the non-shared layers and the different replicas eventually diverge. The reverse transition requires more work and, in the general case, is not possible without application-specific information. In our case, we employ the layering information and the property specifications and handlers to achieve this transition.

Suppose that we want to switch form *graph* sharing to *window* sharing. Initially, users may have completely different views of the shared graph, however, before we switch to window sharing, they must become *identical*. To accomplish this, we walk the tree of layer dependencies (Figure 5.8) in a depth-first manner starting with the new shared layer and, at each step, synchronize the state of the respective layer. For the *GraphDraw* application, one possible layer walk is *figures, graph, graph view, appearance*. The layer synchronization involves finding all registered shared objects that belong to the layer, finding their corresponding properties with their *read* handlers, reading the current state, and sending it as a set of committed *overwrite* operations to remote users. We use the state of the user requesting the sharing mode switch as the master copy with respect to which all other replicas are made consistent. Once dependent layers become consistent, the infrastructure is ready to start window sharing.

5.3.3 Development Costs

The development costs related to adding collaborative functions to the *GraphDraw* application fall into two categories—*XML* declarations and *Java* code. Let us first examine the *XML* specification effort, which consists primarily of property specifications.

As it turns out, *GEF* employs only three basic patterns in the object structures we want to share—standard simple (*JavaBeans*) properties, as well as two versions of *set* properties to maintain a variety of list structures. A *set* property resembles a sequence, however, it has no explicit methods to control the ordering of its elements as illustrated by the following piece of code:

```
public void add(Fig fig);
public void remove(Fig fig);
public Vector getFigs();
```

The actual specifications are simplified versions of the sequence property definitions and are given in the Appendix. The corresponding property handlers are also reused with nominal changes. The layer specifications are also quite simple, owing to the fact that all objects belonging to the *figures* layer and the *graph* layers are subclasses of the `Fig` and `NetPrimitive` classes defined by *GEF*. Hence, a single definition for these classes covers all the different objects that can belong to them.

Table 5.1 summarizes the *Java* coding effort to achieve the described collaboration behavior for all peer-to-peer sharing scenarios for the *GraphDraw* application. To build a centralized version, additional effort would be required to decouple the abstract graph representation from its user interface. While in *GEF* these are separated in different classes, the architecture is not designed for dynamic composition. Therefore, untangling the graph objects and their user interface requires redesign effort, which we consider beyond the scope of our work. One simple way around this problem is to run an extra copy of the existing application version (with the user interface) that is not used by any user with the appropriate commit-based sharing policy.

	GraphDraw	*GEF*		*Total*
Action	*Modified*	*Modified*	*Added*	
Total Lines of Code	26	76	102	204
Affected Classes	5	13	4	22
Code Complexity				
`import` statements	11	15	12	38
`if` statements	0	3	1	4
`for` statements	0	5	1	6

Table 5.1 Code Statistics for *GraphDraw* Application

Let us now consider in more detail the breakdown of the numbers in Table 5.1. We have broken all statistics into two basic categories related to the *GEF* infrastructure, which would be reusable with any *GEF*-based application, and changes to the example *GraphDraw* application. As the numbers suggest, the bulk of the effort is concentrated on adapting and extending *GEF*. The first line gives the sum total of the lines of code that have been added/modified (all changes were tagged at the time of introduction to enable the collection of this data). The *affected classes* row gives the number of classes in which at least one

change related to collaboration has been made (we exclude bug fixes to the original implementation). In the last three lines of the table we provide some basic measures of the complexity of changes introduced. Approximately 20% of all changes are `import` statements, which exist solely for convenience reasons and do not represent a computation. The rest of the effort is fairly straightforward linear code—mostly variable declarations and invocations of infrastructure services—with the exception of four `if` and six `for` statements.

The *added* infrastructure code consists primarily of three event adapters that translate *GEF*-defined events into property and synchronization events. Although this code nominally represents more than 50% of the changes to *GEF*, its implementation is largely routine, as illustrated by the following typical method:

```
public void edgeAdded( GraphEvent e) {
   PropertyOperation po = new PropertyOperation( modelGID, "Edges",
                                                 "add", e.getArg());
   coupler.dispatch( po);
}
```

Thus, the overall coding effort in terms of lines of code represents less than 1% of the original *GEF* code, which supports the automation and reuse claims of our thesis. In our experience, a much greater effort was expended on reverse engineering the design of *GEF* and fixing some of its bugs (the code was a work-in-progress version) rather than adapting it to collaboration.

5.4 Shapes Application

In this section we briefly describe an earlier experience with creating a collaborative version of the *Shapes* application, which was developed for teaching purposes at *UNC*. As Figure 5.13 shows, the application is a drawing editor featuring the customary commands of such applications. It is a simpler application than the *GraphDraw*, however, not a trivial one—it consists of 76 classes and almost 5000 lines of code.

Our primary goal was the validation of our object diff-ing technique and, therefore, we applied diff-based sharing to the main object structure of the application, which consists of a `Hashtable` of `Shape` objects. The original results, featuring a work-in-progress version of our infrastructure, were reported in [21]. Since much of the infrastructure has changed, we provide an updated accounting of the development effort.

For this application, we used an application trigger, which listened for *any* user command and in response triggered a diff on the `Hashtable` by issuing a synchronization event. The coding of this application trigger took about 15 lines of code. In addition, we needed an additional 8 lines in the startup code (very similar to the ones already described for the other applications), 4 additional lines to make the shape table and all `Shape` objects `Identifiable` (by extending `ReplicatedObject`), and 5 additional lines of code to provide global names to all shared application objects. Since the shared table used the standard *table* pattern we did not have to add any property specifications or handler code. Thus, our overall effort was capped at 25 lines of additional code.

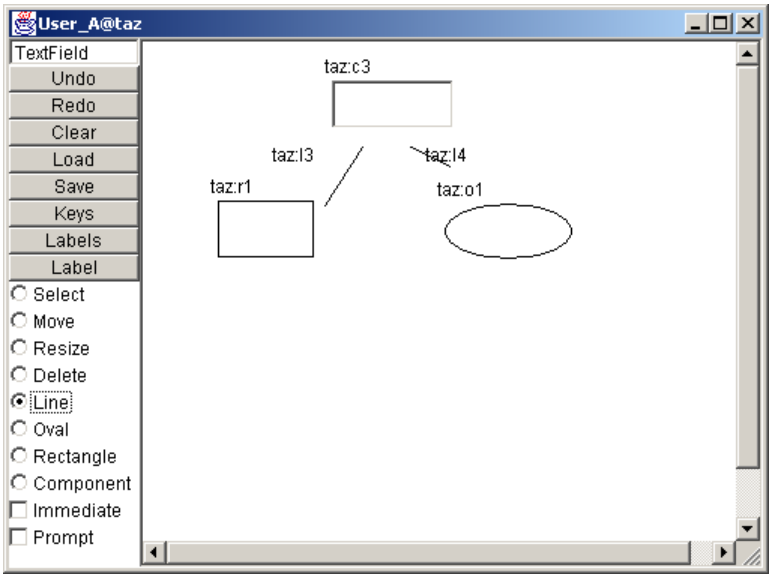


Figure 5.13 *Shapes* application

So far, the driving problem behind our effort has been the implementation of flexible object sharing. In the following sections we look at the utility of the mechanisms that we have developed by applying these mechanism to a set of problem that are unrelated to collaboration.

5.5 XML-Based Object Serialization

Our first problem was originally motivated by the needs of our infrastructure to store various objects, such as sharing policies, persistently. In that respect, *Java* seems to provide an excellent solution through its generic *Java Serialization* mechanism. The programmer's effort is reduced to tagging the object as `Serializable` (that is, the object implements an

empty interface called `java.io.Serializable`) and executing the `writeObject` method of the `ObjectOutputStream`, which represents the persistent store. In addition to the object itself, the serialization will also store copies of any dependent objects to which the serialized object holds references and will recursively apply the serialization algorithm to the dependent objects until an entire self-contained object structure is serialized.

While the outlined mechanism is very convenient, it also suffers from at least two major problems that limit its applicability. The first one we call the *fragile serialization* problem. Since the serialization stores the internal physical structure of the object, any changes to it, such as adding a local variable, would change the physical layout of the local variables, which, in turn, would render the serialized state of the old version unusable. In some situations, the serialization can break even without recompiling code if in the serialization process some execution environment state is also stored: upon deserialization in a different environment, some of the resources that the object refers to may not exist, thereby rendering the whole serialized object useless.

The second problem is that *Java* stores its objects in a proprietary format. Hence, the data is bound to its *Java* object presentation and cannot be used autonomously. Thus, if the serialization breaks, the stored data would be irrecoverable, which largely contradicts the idea of reliable persistent storage.

With the emergence of *XML* as a universal standard for storing structured data, the obvious choice for resolving the proprietary format issue is to replace it with *XML*. However, as demonstrated by *KOML*[11], simply changing the encoding while still relying on the physical object layout does nothing to solve the fragility problem.

Currently, there are two basic approaches to solving the fragility problem. The first one (most widely used) is to shift the burden back on the shoulders of the developers by, essentially, requiring them to implement the *XML* serialization manually. This approach is most prominently illustrated by the standardization effort[12] currently under way, called *Java data binding*, that seeks to define a standard API for mapping *Java* objects to *XML*. While this standardization would be helpful, it would still leave the implementation in the hands of

[11] `http://koala.ilog.fr/koml`

[12] `http://java.sun.com/aboutJava/communityprocess/jsr`

the application developer. Thus, we have a mechanism with maximum flexibility but, unlike the original *Java* object serialization, virtually no automation.

The second approach, exemplified by *KBML*[13], is to extract the logical structure of the object based on *JavaBeans* properties. This allows the serialization to be completed automatically and to store the data in *XML* format that is independent of the serialized object, which solves the fragility problem. The main issue here is that the *JavaBeans* model is very limited in its ability to describe programmer-defined abstractions (this issues was already discussed in the context of object sharing).

Our solution is can be viewed as extending the *KBML* approach by utilizing our pattern-based abstraction model to provide automatic *XML* serialization to a wider range of abstractions than currently possible. We also rely on our architectural model to obtain an extensible solution that promotes code reuse (the implementation of the serialized object remains untouched).

The actual service, called `XMLSerializer`, follows the same generic `ObjectWalk` algorithm we first presented in Chapter 3. The only minor difference is that, at each step, the processing (i.e., *XML* encoding) of an object is split into three parts—generation of opening *XML* tag, recursive serialization of properties, and generation of closing *XML* tag—instead of two shown in the generic version:

```
XMLSerializer(object)
  if object == null || object in visited list
    return
  <add object to visited list>
  <start XML encoding of object>
  for each property p_i of object do
    reader = lookup read handler for p_i
    if reader != null
       values[] = reader.getValue(object,pi)
       for each v in values
         ObjectWalk(v)
       end
    fi
  end
  <complete XML encoding of object> // extra processing
end
```

Note that, in this case, we need only the standard *read* handlers to complete the task because the only operation we need from the property is to obtain its current state. Thus, any

[13] `http://koala.ilog.fr/kbml/`

object property defined through our specification mechanism is automatically *XML* serializable.

To illustrate the work of our service consider the serialization of the example object structure shown graphically on Figure 5.14. On the diagram, rectangles represent objects, whereas rounded rectangles represent object properties. The generated *XML* representation is shown on Figure 5.15. It is worth noting that the end result is an *XML* document that is completely independent of the original *Java* object. The only reference to it is the `javaClass` attribute, which is the minimum information we need to enable the reverse deserialization process.

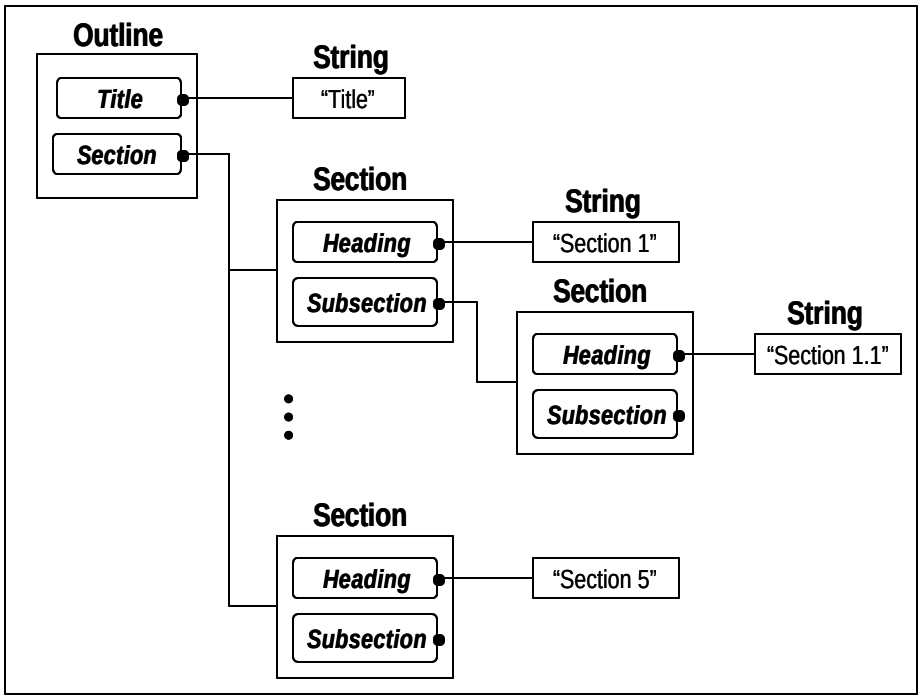


Figure 5.14 Example Outline object for *XML* serialization

```
<?xml version="1.0" encoding="UTF-8" ?>
<outline javaClass = “outline.Outline”>
  <title>Title</title>
  <sections>
    <section javaClass = “outline.Section”>
       <heading>Section 1</heading>
       <subsections>
         <section javaClass = “outline.Section”>
           <heading>Section 1.1</heading>
           <subsections/>
         </section>
       </subsections>
    </section>
    ...
    <section javaClass = “outline.Section”>
      <heading>Section 5</heading>
      <subsections/>
    </section>
  </sections>
</outline>
```

Figure 5.15 *XML* serialization of example object

In summary, our outlined *XML* serialization implementation has the following properties:

Automation. Once the set of properties and respective handlers is defined, the conversion is done automatically and requires no further intervention. As already discussed, application programmers tend to use a relatively small number of patterns relative to the number of classes in an application. By defining the serialization on a per-pattern basis, rather than per-class basis, the developer could realize significant savings, as well as simplify the management of any changes to the conversion scheme. Also, patterns are likely to change little from application to application, opening the possibility for complete reuse of the serialization handlers.

Generality. The conversion scheme can be applies to an extensible set of objects through the addition of property specifications. In particular, unlike other data binding, it is independent of any particular API. Furthermore, it can cover a wider range of abstractions than *JavaBeans*-based schemes.

Stability. Our approach solves the fragile serialization problem in a more general case than existing systems

Reversibility. Using an analogous recursive procedure, the reverse *XML*-to-object conversion can also be executed, thereby providing support for the complete serialization/deserializatrion cycle.

5.6 UPnP Device Interoperation

Let us now consider a problem that arises in the context of device interoperation. To enable such interoperation, a number of (sometimes competing) protocols, such as *UPnP*[14] (Universal Plug-and-Play), *Jini*[15] (for *Java* objects), and *WSDL*[16](Web Services Description Language). From the point of view of a device developer, this presents the problem of having to maintain a number of external interfaces that are not directly relevant to the device's functions and, therefore, present development overhead. For example, a weather station device can be exported as a *UPnP* device, a *Jini* device (if its implemented in *Java*), or it could be treated as a web service and be exported using *WSDL* (Figure 5.16).

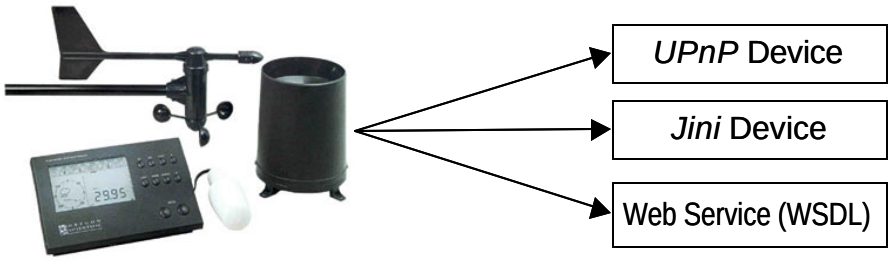


Figure 5.16 Possible standardized interfaces for a weather device

Ideally, it should be possible for the developer to focus entirely on developing and improving the device and have the external descriptions automatically generated. Unfortunately, to the best of our knowledge, no such tools currently exist. In fact, the tools that are in use take the exact opposite approach—they start with the respective device specification and generate a skeleton programming language implementation, which the developer must fill in. While it is usually straightforward to implement such an approach, it carries at least two non-trivial limitations.

The first one is that each specialized tool takes care of specific protocol and it is unlikely that two separate tools will generate compatible skeleton implementations. The second one is that, over the lifecycle of a device, its description is likely to change as new versions are

[14] `http://upnp.org`

[15] `http://jini.org`

[16] `http://www.w3.org/TR/wsdl`

developed. Existing approaches poorly support this process because they force the developer to come up with a new device description, generate a new skeleton, and again fill in the code.

Clearly, both of the above problems stem from the fact that (skeleton) device code is generated based on the external description, whereas what is truly needed is support for the reverse process—generation of an external description based on the native device implementation. This would allow the developer to focus on the device and use protocol-specific tools to automatically deal with the external protocol representation.

In this section, we present our experience in generating external descriptions of *Java*-based devices using the *UPnP* protocol. *UPnP* uses *XML*-encoded device descriptions and commands communicated via HTTP to facilitate device interoperation. The main obligations of a *UPnP* can be summarized as follows:

- *Export XML descriptions of services and actions*. Generally, it is expected for *UPnP* services to be rather simple. To illustrate, consider the simulated *StereoDevice* shown below (Figure 5.17). In this case, the device might define four services: *power* management, *input* selection, *track* navigation, and *volume* (control). The actions, for the power service, for example, could be *getPower* (obtain the current power status) and *setPower* (change the power setting, turning the device on/off). For the track navigation, possible actions include *getTrack*/ *setTrack*, but also *nextTrack*/ *previousTrack*.

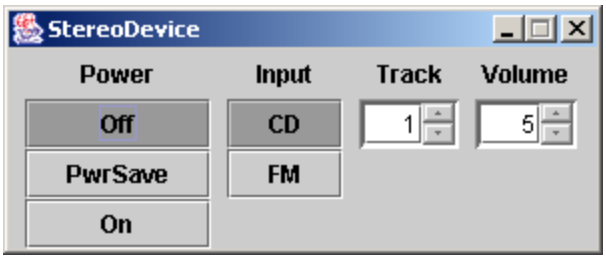


Figure 5.17 Simulated *StereoDevice*

- *Provide HTML-based user interface*. Each device is expected to provide an HTML interface to observe and control its status.
- *Generate events.* A device may generate events announcing changes in its state that other devices can subscribe to receive.

Taken together, the above *UPnP* requirements can incur significant development overhead, especially for a device undergoing active development with frequent changes.

Therefore, the goal of our work has been to automate the process by providing automatic support for all of the above requirements. In this section, we present our experience in addressing the first two requirements and sketch ideas on how to extend support to the third one.

The basic design is based on the idea of providing a *UPnP* proxy service, which automatically fulfills the *UPnP* obligations on behalf of native devices (Figure 5.18) and in the following sections we successively describe our pattern-based approach to providing such a proxy service for *Java* objects.

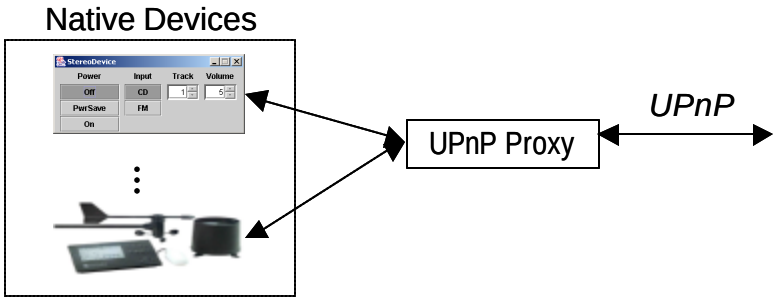


Figure 5.18 Exporting devices through a *UPnP* proxy

5.6.1 System Interface Generation

The essential idea of our design is to use properties to define (*UPnP*) services and pattern methods to define actions. Our actual implementation relies on the *Java UPnP Stack*[17] implementation from *Siemens* to handle the low-level details of the protocol. The *UPnP Stack* provides a high-level object API and handles, for example, the actual cumbersome *XML* generation of device/service description. However, the programmer is still responsible for defining the device, services, and actions in using the system-provided *Device*, *Service*, and *Action* classes, respectively.

To make our discussion more concrete, let us use the simulated *StereoDevice* presented above to illustrate our discussion. In particular, we use the following device implementation for that purpose:

```
public class StereoDevice {
    public static final String[] POWER_VALUES = {"Off","PwrSave","On"};
    public static final String[] INPUT_VALUES = {"CD","FM"};
    public static final int[] VOLUME_RANGE = {0, 10};
```

[17] http://www.plug-n-play-technologies.com

```
    public String getPower();
    public void setPower( String input);
    public String getInput();
    public void setInput( String input);
    public int  getTrack();
    public void setTrack( int track);
    public void nextTrack();
    public void prevTrack();
    public int  getTrackMin();
    public int  getTrackMax();
    public int  getVolume();
    public void setVolume( int volume);
    public void volumeUp();
    public void volumeDown();
}
```

The first task of every device is to announce its presence and advertise its services. As already mentioned, we use properties as a guide to define services and we map each object property to a device service. The entire process is illustrated by Figure 5.19, where each numbered arrow represents a step in the process and is explained below in detail.

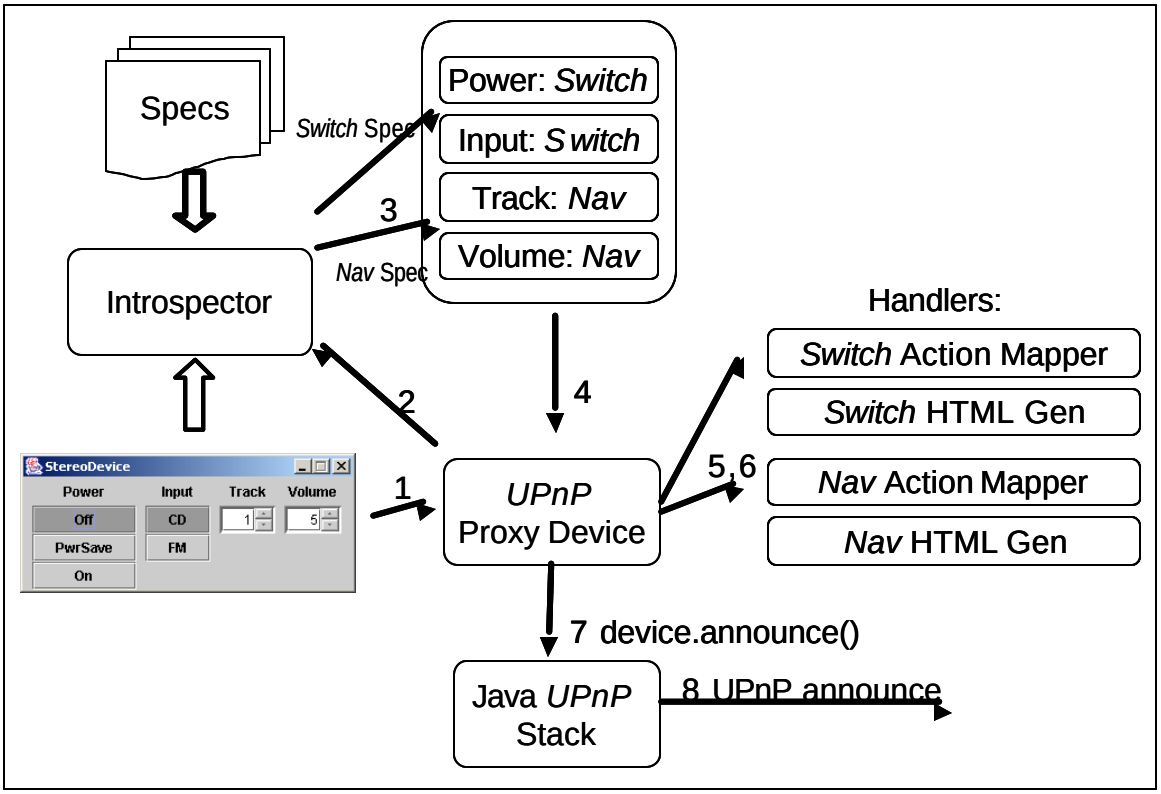


Figure 5.19 Announcing a *UPnP* device through a proxy

1. Our starting point is an instance of the device object (`StereoDevice`), which is created by the proxy.

2. Since the proxy has no knowledge of any particular device, it hands the device object to the `Introspector` for property analysis, which is based on a set of provided specifications. In the example, we define two types of properties: *switch* and *navigation* (referred to in diagrams as *nav*). These are both versions of the simple properties we used before. The difference is that a *switch* property is of `String` type and can take one of a fixed set of values (the name refers to the fact that is used to model switches that users manipulate). *Navigation* properties are distinct from the simple ones in that they are of integer type and define additional increment and decrement methods.
3. Based on these definitions (the actual *XML* definitions are almost identical to the *simple* property definitions), the `Introspector` decomposes the object into a set of properties.
4. Given a set of properties, the proxy defines a service for each discovered property. There are two *switch* properties (*power* and *input*) and two *navigation* properties (*track* and *volume*) in our `StereoDevice` object for which services are defined.
5. Next, the proxy must map (that is, encode in *XML*) the actions of service, which correspond to our pattern methods. Such encoding is inherently property method-specific and, therefore, is not embedded into the proxy. Instead, an `action_mapper` property handler is defined for each property that performs this task. The proxy uses the specification to lookup and invoke the respective handler for each of the *switch* properties.
6. Similarly, the proxy uses the specification to lookup and invoke the respective handler for each of the *navigation* properties.
7. Once the object encoding of the device is completed, it is handed off the *UPnP stack* with an announcement request.
8. The *UPnP stack* announces the device to the world using the *UPnP* protocol (it will also remember this announcement and present it to any future inquiries it might receive).

5.6.2 User Interface Generation

As part of the announcement, the device must include a *URL* at which an HTML user interface for interacting with the device must be specified. Again, this is a task that, from the point of view of the device developer, is pure overhead and should also be automated. To the best of our knowledge, current *UPnP* toolkits provide little in the way of automating this task, largely because they have no means of interpreting the device services/actions in a generic way. In contrast, we employ property-specific handlers to enable the automatic generation of the HTML interface. We use the following diagram to explain our solution step by step. Note that, at this point, the property analysis has been completed and the device has been announced.

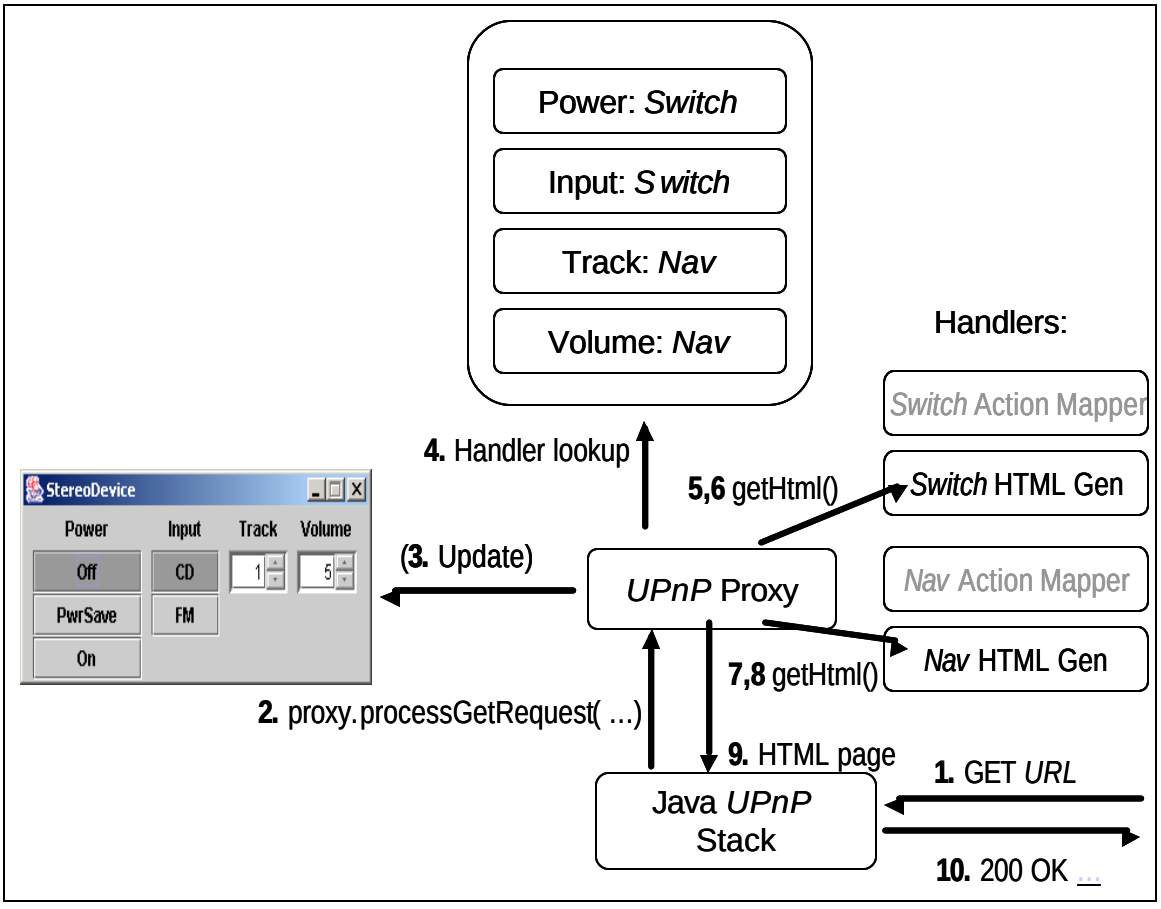


Figure 5.20 Servicing an HTTP request

1. The *UPnP stack* receives a regular HTTP GET request from a client.
2. The URL of the request is passed on to the proxies, which has registered at startup as the handler for such requests.

3. The proxy parses the URL and executes any commands that might have been encoded (this is syntactic process and can be applied to any object).
4. The proxy looks up the (cached) list of properties for the device object.
5.– 8. Based on the specifications, it consecutively looks up and invokes the corresponding html_generator property handler for each of the four properties. Since we have two types of properties, there are two actual handlers defined.
9. Combining the HTML produced by the handlers with some wrapping code for the whole device, the proxy returns the HTML page to the stack.
10. The UPnP stack responds to the client request.

5.6.3 *UPnP* Summary

Above, we presented the implementation of a *UPnP* proxy service, which uses the developed pattern-based mechanisms to automate the process of exporting a native device as a *UPnP* device. Our solution has several notable properties:

Device-centric. To the best of our knowledge, no other existing tool provides support for automatically exporting a device. As already discussed, this better supports the development cycle and leaves open the possibility to generate multiple (independent) external system interfaces, through which the device is accessible.

Generic. The proxy can work for any device object, as long as it has defined properties. Note that, at no point does our solution require changes to the original device implementation and the addition of new devices does not require changes to the proxy.

Extensible and customizable. The set of recognizable properties and, hence, device services can be extended by adding more property definitions and respective handlers without recompiling device or proxy code. Moreover, by swapping property handlers the system can be incrementally customized, e.g., change the HTML interface, or provide multiple versions of it.

Code reuse. In addition to fully reusing device and proxy code, our infrastructure allows handler code to be readily reused across devices (e.g., multiple devices are likely to have switch/navigation properties and they can share the HTML interface).

Finally, we would like to point out that our diff-ing service could be used to deduce and announce changes on behalf of a device that does not, by default, issue events.

5.7 Summary

In this chapter we shared part of our experience in developing multi-user application using our prototype implementation, as well as presented our experience with non-collaboration problems. Through this effort we sought to demonstrate that the design rationale of our conceptual model is supported in practice. Our primary focus has been on measuring the implementation effort in order to quantify the infrastructure's achievements with respect to automation and code reuse.

Our results show that we were able to achieve a level of sharing flexibility that is as high as that of our benchmark systems—*Suite*—at the cost of less than 1% of the original single-user code for a non-trivial piece of software, such as *GEF*. Furthermore, for the special case of WYSIWIS sharing we require no specific collaboration support from the application just like our benchmark system *JCE*.

With respect to code reuse, we should point out that virtually all modifications to the original code were simple additions and did not require changes the logic of the single-user code. Furthermore, our mechanisms are designed to be complimentary to the basic reuse mechanism in object-oriented programming—class inheritance. That is, in addition to sharing code with their superclasses, by default, subclasses also share the *XML*-based specifications and respective property handler code. This feature is a direct result of the fact that patterns, as a mechanism to describe the object's logical structures, completely subsumes inheritance.

6. EVALUATION

The goal of this research has been the design and implementation of a collaborative infrastructure that *better* satisfies the generic infrastructure requirements of automation, flexibility, extensibility, and code reuse, as defined in Chapter 1. Recall that by better we mean that for a given a level support for three of the requirements, our infrastructure will exceed, or at least match, the achievements of the benchmark systems with respect to the fourth requirement (in some cases, existing solutions are optimal so matching them is the best we can hope for). The results of our evaluation are summarized in a evaluation tables, which rank each surveyed system with respect to each of the infrastructures requirements.

6.1 Method of Evaluation

We have used two primary methods of evaluating the sharing infrastructures discussed in this dissertation with respect to the requirements—inspection and simulation. Inspection refers to the use of objective observations of the design and implementation of each system to establish the degree to which a certain requirement is met. For example, by examining the design and usage of an infrastructure, we can answer the question of whether or not the system supports compiled code reuse.

To establish comparative results between a target infrastructure and our own we use simulation as our primary tool. That is, we compare the original implementation of a reference application using a particular infrastructure, with an analogous implementation that uses our infrastructure and simulates the features of the given infrastructure. As already discussed in Chapter 1, the basic criterion upon which we draw conclusions is the following: if our infrastructure can simulate the main (conceptual) features of another infrastructure with respect to a particular requirement with a comparable amount of effort, we consider our infrastructure to be at least as good as the other. If, in addition, we support features not present in the other infrastructure, we consider ours to be better with respect to the requirement.

In practical terms, providing complete implementation simulations for all of the surveyed systems is not feasible for at least two reasons. First, the different infrastructures are based on different software platforms that may exhibit features that are simply not available to our *Java* implementation. Second, the sheer implementation effort puts such a study beyond the scope of this dissertation. Therefore, our approach is to simulate several of the systems, an effort already described in the previous chapter, and provide an implementation sketch for the rest of the systems.

Let us now describe the scoring scale for each of the requirements, as well as the criteria used to assign the specific scores. A more detailed explanation of individual scores is given after the tables.

- *Automation:* Low/Moderate/High/Complete. We rate the automation of a system as *Low* if the effort to implement collaboration using the infrastructure becomes comparable to that of implementing it without the infrastructure. This may be due either to the low level of shared abstraction, which would require the developer to build higher-level abstractions, or the assumption of a very specific programming model that would essentially require the redesign of most applications.

  A *Moderate* designation means that the implementation effort requires some non-trivial mapping between the application's shared structures and the shared abstractions provided by the infrastructure. After this mapping is established, however, the infrastructure takes care of all the implementation details from that point on.
  By *High* automation, we mean that the needs of a wide range of applications can be directly accommodated by the shared abstractions provided by the infrastructure and, as a result, the implementation effort for the multi-user application version is comparable to that of the single-user version. The programmer is required to provide some basic information and, perhaps, run some specialized initialization code but, by and large, no serious changes are required to implement collaboration.

  Complete is the highest rating and means that the single- and multi-user versions of the application code are identical. As the following tables suggest, this can only be achieved in a limited number of scenarios.

- *Sharing flexibility:*
  - *Concurrent updates:* Yes/No. With respect to this requirement, we ask whether the infrastructure permits concurrent updates to a shared object by multiple users.
  - *Semantic sharing:* Yes/No. We ask whether the infrastructure supports the independent sharing of a semantic object.
  - *User interface sharing:* Yes/No. We ask whether the infrastructure supports the sharing of the user interface of the application.
  - *Sharing modes:* Sync/Async/Flexible. The score with respect to this category describes (at a very high level) the types of sharing modes supported by the infrastructure. The *Sync* and *Async* values correspond to (pure) synchronous and asynchronous sharing, while an entry of *Flexible* indicates that the infrastructure also support some intermediate forms either based on parameters (e.g., *Suite, TACT*), or fixed points (e.g., *JViews*).
- *Abstraction flexibility:*
  - *Programmer-defined semantic objects:* Any/No/<Object description>. With respect to this requirement, we have no fixed-point scale to differentiate among the infrastructures. By *Any* we mean that potentially any object can be turned into a shared object (at a nominal effort), if the sharing mode is fixed. By *No* we mean that it is not possible for the application programmer to define objects that are directly shared by the infrastructure. If neither *Any* nor *No* applies, we briefly describe the range of shareable objects and provide a detailed explanation in the respective narrative section explaining the score.
  - *Programmer-defined user interface:* Yes/No. To evaluate with respect to this requirement, we ask the question: Is it possible for the developer to write a custom user interface and still get sharing from the infrastructure for sharing both the semantics and the user-interface? This category applies only to infrastructures that support UI sharing; those that do not are marked with N/A.
- *Specification flexibility:*
  - *Late specification binding:* Yes/No. With respect to this requirement, we pose the question: Is it possible for the user to specify the sharing mode without recompiling the application?

- *Ease of specification:* Low/Moderate/High. By *Low* we mean that the infrastructure requires a procedural specification of the sharing mode (i.e., writing code), which effectively precludes the user from routinely adjusting it at run time. A *Moderate* designation refers to a system that is parameter-based but its parameters values do not directly relate to the application objects manipulated by the user. Rather, they apply to an abstract model defined by the infrastructure. Therefore, the developer would have to map application abstractions to the model and build a specialized user interface to make these parameters available to the end user in an intuitive form. By *High* we mean that the specification mechanism closely follows the logical structure of the objects seen by the user and the infrastructure provides a generic user interface through which the user can manipulate the sharing parameters. Finally, we should note that we link this category to the previous one in that we only evaluate infrastructures that have a late specification binding mechanism. For those that do not, the requirement is trivially satisfied and, therefore, is marked with *N/A*.

- *Code reuse:*
  - *Compiled code reuse:* Yes/No/Complete. With respect to this requirement, we ask whether it is possible to directly reuse compiled code from the single-user version of the application to build the collaborative one. Apart from the obvious *Yes/No* answers, we also designate with *Complete* all infrastructures in which this kind of reuse is taken to the extreme by reusing the entire application code without recompiling.
  - *Incremental collaboration awareness:* Yes/No. Recall that, by incremental collaboration awareness, we mean the ability of an infrastructure to accommodate the gradual transformation of a regular application into a collaborative one. An infrastructure supporting such incremental approach provides multiple levels of service, depending on the collaboration awareness of the application. Thus, the table entry reflects whether or not this is the case.
- *Extensibility:*
  - *Separation of shared abstractions and their implementation:* Yes/No. The evaluation of this requirement is based on answering the question: Is it possible to compose the application with a completely different sharing implementation without modifying the

application code? Specifically, does the infrastructure design envision and facilitate such a transition.

- *Sharing functions separation:* Yes/No. The evaluation of this requirement is based on answering the question: Is it possible to vary the implementation of one infrastructure function without modifying, or even knowing of, other infrastructure functions?
- *Late component binding:* Yes/No. Here, we pose the question: Is it possible to compose the application and infrastructure services, as well as the different infrastructure services, at run time? Evidently, this question is only valid for infrastructures satisfying at least one of the two preceding requirements; those that do not are marked with *N/A*.

6.2 Evaluation Tables

REQUIREMENTS \ INFRASTRUCTURES	**OUR**	***JCE***	***GroupKit***	***Colab***	***Suite***	***JViews***	***DISCIPLE***
AUTOMATION	Complete/ High	Complete	High/High	High	High	High	High
FLEXIBILITY							
Sharing flexibility							
Concurrent updates	Yes	No	Yes/Yes	Yes	Yes	Yes	Yes
Semantic sharing	Yes	No	Yes/Yes	Yes	Yes	Yes	Yes
UI sharing	Yes	Yes	No/No	No	Yes	Yes	Yes
Sharing modes	Flexible	Sync	Sync/Sync	Sync	Flexible	Flexible	Sync
Abstraction Flexibility							
Programmer-defined semantic objects	Any/ Pattern-based	Any	No/Any	Any	Concrete types	*JViews* objects	*JavaBeans*
Programmer-defined user interface	Yes	Yes	N/A	N/A	No	Yes	Yes
Specification flexibility							
Late specification binding	Yes	No	No	No	Yes	Yes	Yes
Ease of specification	High	N/A	N/A	N/A	High	High	Low
CODE REUSE							
Compiled code reuse	Complete/ Yes	Complete	No/No	No	No	Yes	Yes
Incremental collaboration awareness	Yes	No	Yes/Yes	Yes	No	No	No
EXTENSIBILITY							
Shared abstraction/implementation separation	Yes	No	No/No	No	No	Yes	Yes
Sharing functions separation	Yes	No	No/No	No	No	No	Yes
Late component binding	Yes	N/A	N/A	N/A	N/A	Yes	Yes
Simulatable	N/A	Yes	Yes/Yes	Yes*	Yes*	Yes*	Yes

Table 6.1 Evaluation: Infrastructures vs Requirements (Part 1)

REQUIREMENTS \ INFRASTRUCTURES	**OUR**	***AMF-C***	***DFS***	***Coda***	***Notes***	***Bayou***	***TACT***	***Sync***
AUTOMATION	Complete/ High	Low	Complete	Complete/ Low	High	Low	Moderate	High
FLEXIBILITY								
Sharing flexibility								
Concurrent updates	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes
Semantic sharing	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
UI sharing	Yes	Yes	No	No	No	No	No	No
Sharing modes	Flexible	Flexible	Async	Async	Async	Async	Flexible	Async, Sync
Abstraction Flexibility								
Programmer-defined semantic objects	Any/ Pattern-based	*AMF* objects	Any	Any	*Notes Objects*	Database Records	Database Records	*Replicated* Objects
Programmer-defined user interface	Yes	Yes	N/A	N/A	N/A	N/A	N/A	N/A
Specification flexibility								
Late specification binding	Yes	Yes	No	No/ Yes	Yes	Yes	Yes	Yes
Ease of specification	High	Low	N/A	N/A/ Low	High	Low	Moderate	High
CODE REUSE								
Compiled code reuse	Complete/ Yes	No	Complete	Complete	No	No	No	No
Incremental collaboration awareness	Yes	No	No	Yes	No	Yes	Yes	Yes
EXTENSIBILITY								
Shared abstraction/implementation separation	Yes	Yes	No	Yes	No	Yes	No	No
Sharing functions separation	Yes	No	No	No	No	No	No	No
Late component binding	Yes	Yes	N/A	Yes	N/A	Yes	N/A	N/A
Simulatable	N/A	No	Yes	Yes	Yes*	Yes	No*	Yes

Table 6.2 Evaluation: Infrastructures vs Requirements (Part 2)

6.3 Comparative and Absolute Evaluation of Each System

In the following sections, we evaluate each of the systems in the survey, as well as our own, and explain the specific tables entries that we have given for each system. Generally, the narration for each of the systems follows the order of the requirements in the above evaluation tables. We also perform a comparative evaluation of our work relative to the benchmarks to show that we have met our original design goals.

6.3.1 JCE

Recall that *JCE* is one of our benchmark systems with respect to automation and code reuse. Like other user interface-based systems described in Section 2.1, it uses the sharing of the user interface representation of an object to share the object itself.

Based on this model, it provides fully automatic WYSIWIS sharing of arbitrary *Java* applications and, therefore, *Complete* automation. Since *JCE* has no notion of the underlying shared semantic object defined by the application, it disallows concurrent updates and cannot support separate sharing of the semantic object. At the same time, the WYSIWIS model implies *synchronous* (only) sharing of the *user interface*.

With respect to abstraction flexibility, *JCE*'s model places no restrictions on the user interface implementation or the underlying semantic object. With respect to specification flexibility, none is supported in that there is a single sharing mode, which in turn implies no (need for) late binding mechanism.

As already discussed, the compiled code reuse requirement is completely satisfied, however, incremental awareness is not in that the application cannot help improve the sharing support it gets provided by the infrastructure. Moreover, the infrastructure itself was not designed to be extensible in any way so that its sharing support can be improved upon.

As our discussion on Section 5.2 showed, we were able to simulate *JCE*'s model for *Java Swing*-based applications by sharing the input/output event queues of the application. The *Swing* library is a second-generation implementation of the standard *Java* user interface toolkit designed to replace the first generation *AWT*. We did not implement sharing of *Java AWT*-based user interfaces and the reason for this is purely technical—the *AWT* implementation is closely tied to the underlying window system and sharing it required numerous modifications to the standard *AWT* library to gain complete control of the event

streams. Since such modification would be irrelevant to our primary goals, and would be largely repeating the solutions employed by *JCE*, we have opted not to include them as part of our infrastructure.

6.3.2 GroupKit

In the case of *GroupKit*, we have an infrastructure that provides two distinct sharing mechanisms—*shared environments* and *multicast RPC*—and our evaluation accounts for that by giving two corresponding scores. With respect to multicast RPC, we consider only the synchronous version because it can be directly used to implement object sharing. The asynchronous version we consider a general-purpose multicast mechanism that requires non-trivial additional design and programming effort to build a shared abstraction.

Since the programmer can directly employ both of *GroupKit*'s basic mechanisms, overall, the system provides a *High* level of automation. The system supports concurrent updates and it appears that, absent an application-defined coordination mechanism, the exact results depends on how the communication mechanism serializes the sequence of operations. Both shared environments and multicast RPC were designed with the idea of semantic object sharing and it was assumed that achieving the desired level of UI consistency is the responsibility of the developer. Hence, despite the fact that *GroupKit* provides collaborative widgets, such as telepointers and multi-user scrollbars, it does not provide an automatic mechanism for sharing the application's UI.

With respect to abstraction flexibility, there is no mechanism that would permit the definition of programmer-defined abstractions, other than the use of multicast RPC, which can potentially be applied to *Any* object. Because of its exclusive focus on synchronous sharing, *GroupKit* provides *No* sharing flexibility.

Compiled code reuse was not a design consideration for *GroupKit* and, hence, no specific support is provided. In contrast, it is possible to take an application and to gradually add collaborative features to it adding multicast RPC invocations on select objects. Finally, there are no specific design features that allow the infrastructure services to be easily extended or replaced by the developer.

Environment sharing is a special case of the sharing provided by our infrastructure and can be readily simulated. For example, one way to implement a shared environment is to

subclass *Java*'s standard `Hashtable` class and add a notification mechanism to enable synchronous sharing as sketched below:

```
public class SharedEnvironment extends Hashtable implements Identifiable {
  Coupler coupler = CentralRegistry.lookup( <couplerName>);
  // Identifiable implementation
  public GID getGID() {…}
  public void setGID(GID gid) {…}
  // Override modifier methods
  public Object put(Object key, Object value) {
    super.put(key,value);
    coupler.dispatch( new PropertyOperation(…));
  }
  public Object remove(Object key) {…}
  …
}
```

By default, our infrastructure supports the sharing of *table* properties and has a property specification that matches the pattern used in the Hashtable class. Hence, with the exception of a registration call upon object instantiation, the above few lines of code are everything needed to implement shared environments. Furthermore, our implementation would get support for other sharing modes at no additional effort. A simulation of the multicast RPC mechanism is provided for object-oriented context in the following section.

6.3.3 Colab

Colab is another one of our benchmark systems. In essence, it provides synchronous sharing for arbitrary objects (subject to some nominal restrictions) by automatically performing invocations of *broadcast* methods on all replicas. Hence, the emphasis is on automation with virtually no flexibility. The specific table entries here are identical to those for *GroupKit*'s multicast RPC (the values after the slash) which reflects the fact that two mechanisms are analogous and the same reasoning applies in justifying the table entries.

With respect to simulating *Colab*'s broadcast methods, our infrastructure provides a `RemoteMethodCall` object, which encapsulates a method invocation that can be replicated on a remote host. The difference between our mechanism and *Colab*'s is that the latter is a compile-time mechanism, whereas ours is not. Consequently, the application object must explicitly construct and send the `RemoteMethodCall` object (1-2 lines of code per method). Certainly, this requires a bit more effort, however, compiler-support options do not appear very appealing. Two points are relevant here.

First, since we cannot legally add a new keyword in *Java*, compiler support would essentially mean a precompiler, which replaces either a new keyword (e.g., *broadcast*) or special comments (a.k.a coding between the lines) with generated code. In both cases, the resulting code violates the basic principle (if not the technical definition) of *Java* as a standardized language. Second, tagging broadcast methods alone is not sufficient when implementing *flexible* sharing because the semantics of each method and its relationships with other methods remain unknown.

We should also point out that our infrastructure allows broadcast methods to be defined in multiple layers of the application. In *Colab*, broadcast methods must be restricted to one layer because otherwise replicas may get multiple executions (in [22] the authors describe an informal methodology for selecting broadcast methods, which roughly translates into identifying modifier methods of shared semantic objects). However, by treating broadcast calls like property operations and using the application layering information, we can suppress causally related multiple notifications and maintain correctness.

6.3.4 Suite

Suite is our most important benchmark system with respect to automation, sharing flexibility, and specification flexibility. Among the surveyed systems, it is the only one that satisfies all aspects of these requirements. In short, *Suite* has the most comprehensive sharing model and automatically supports the widest range of sharing scenarios through its parameterized coupling model. It coordinates the simultaneous input of multiple users and addresses the need for flexibility in both semantic and UI object sharing. The hierarchical specification model provides flexible, run-time specifications at various levels of granularity that can easily be specified by the user.

. Since our sharing and specification models, described in Chapter 3, have been built as proper extensions of those of *Suite,* the simulation of the latter is trivial. The only major difference is that we have extended its essential approach from sharing of concrete data types to an extensible set of programmer-defined abstract data types. Let us briefly consider the mapping of *Suite*'s shared entities to our own.

Variables. We map these to *JavaBeans simple* properties with the essential read and write operations performed by the *getter/setter* methods, respectively.

Records. By decomposing an object into a static set of properties, we can view it as record. In other words, wherever *Suite* uses a record, we can map that to an object with the same set of recognizable properties. Reading/writing the values of each property is performed through *read/write* handlers.

Sequences. Our infrastructure explicitly supports sequence properties as one of the basic property types. Hence, wherever *Suite*'s uses a sequence, we replace that with an object that has a recognizable *sequence* property. The essential insert/delete operations are performed through appropriate *insert/delete* handlers.

Thus, we were able to map *Suite*'s shared structures to our own and were able to implement their sharing without assuming a specialized user interface. Consequently, we claim to have implemented a logically equivalent version of the *Suite* model rather than an exact replica. (Notably, we provide none of the user interface generation facilities upon which *Suite* relies.) The supporting argument was already provided in Chapters 3 and 5, in which discussed the conceptual and implementation issues involved in simulating the *Suite* model.

6.3.5 JViews

JViews shares many of its design objectives with our own. Specifically, it provides shared (*JViews*) objects that support *High* automation, as well as most aspects of sharing flexibility—concurrent updates, separate model/view sharing and two modes of sharing: synchronous and asynchronous (we classify the manual application of updates as asynchronous because it has to be explicitly triggered by the user). I

In terms of abstraction flexibility, *JViews* supports only a fixed set of system-defined abstractions for the semantic objects and does not deal with hierarchical shared structures. However, the basic sharing mechanism, which focused on low-level event flow routing as a means of achieving sharing, is readily applicable to UI objects. The specification model is static providing several fixed sharing policies (one synchronous and several user-mediated versions of asynchronous) that are common to all objects. Hence, late binding of the sharing specification is not supported.

JView takes advantage of *Java*'s late binding mechanisms and supports compiled code reuse of *JViews* and *JavaBeans* objects. However, the low-level event model does not

provide a good basis for gradually introducing collaboration awareness—the developer must understand the whole communication/coordination mechanism before making changes to the original code. Other consequences of this implementation model include shared objects/ sharing mechanisms separation (a shared object issues events that are processed by the infrastructure) and direct dependencies among infrastructure objects implementing different aspects of the event processing. This, in turn, makes separation of function a difficult task.

With respect to the simulation, our infrastructure is capable of sharing any fixed number of system-defined abstractions and, in this sense, capable of implementing *JViews*' sharing model. The approach would be very similar to the effort involved in converting the *GraphDraw* application described in Section 5.3. The main programming effort would be the conversion of the application event model to property operations.

As a matter of design choice, we do not support user-mediated application of remote updates, although it would trivial to do so by simply visualizing the queue of pending property operations. The main reason for this choice is that our goal is to provide some predictable sharing behavior. Once control is relinquished to the user, the system can no longer guarantee the correctness of sharing. This is especially true when switching between the sharing of different application layers.

6.3.6 DISCIPLE

DISCIPLE was primarily designed for synchronous sharing of *JavaBeans* objects and shares many of its characteristics with *JViews*. Thus, instead of repeating the same arguments, we focus on highlighting the differences. The first notable difference is the exclusive support for synchronous sharing, which appears to be a matter of design choice, rather than an inherent limitation. The abstraction model is a subset of that of *JView*'s because, by design, a *JavaBean* is a special case of a *JView* object.. It is possible in DISCIPLE to compose and run the all the application and infrastructure components at run time using a standard visual bean composition tool (called *BeanBox*), which, in principle supports late binding. However, this visual interface was not designed for run-time adjustments to the sharing process. Rather, its intended purpose was to create an application configuration and save it for later use. Therefore, to bend its use for run-time adjustments would require a lot of work on part of the user (hence, the *Low* ease of specification).

The code reuse and extensibility arguments are, essentially, the same as that of *JViews.* The only notable difference is that DISCIPLE features a component-based architecture where different phases of event processing are well-defined, thereby allowing for clean separation of infrastructure functions.

As already discussed, our infrastructure can directly share *JavaBeans* objects. The only effort required on part of the developer is to ensure that shared objects implement the `Identifiable` interface and to attach as a listener the provided `BeanEventAdapter` class that converts bean events into property operation events.

6.3.7 AMF-C

Being a collaborative extension of the single-user *AMF* infrastructure, *AMF-C* was designed to flexibly share objects originally implemented according to the *AMF* architecture. Hence, to a large extent, the system satisfies our sharing and abstraction flexibility. The main problem here is that very few applications are actually developed using *AMF* and, therefore, flexibility comes almost entirely at the expense of additional programming effort. That is, we can expect a non-trivial redesign and re-implementation effort to fit an arbitrary application to the *AMF-C* model. Hence, the *Low* scores for automation and code reuse. Furthermore, because of the low-level event flow specification used to define sharing, the specification effort is considerable, which makes it impractical for manipulation by end users.

With respect extensibility, *AMF-C*'s features are similar to those of *JViews* because both systems employ low-level event processing mechanism to implement sharing. Therefore, while the sharing mechanism is separate from the shared objects, it is difficult to separate different infrastructure service because they directly refer to each other.

We believe that our abstraction model can accommodate *AMF* objects, each of which defines *facets* that, generally, map to properties on our model. However, our sharing model works at a higher level of abstraction (programmer-defined objects) than *AMF-C* counterpart (application-defined events that are opaque to the infrastructure). Hence, while it is possible to simulate lower-level abstractions with higher-level ones, we do not believe that in this case the solution would be contrived.

In conclusion, we should note that *AMF-C* is not one of our benchmarks because it does not provide a high level of automation Recall that our basic requirement is automation and,

therefore, we compare flexibility features of infrastructures that provide high automation, such as ours.

6.3.8 Traditional Distributed File Systems (DFS)

Traditional DFS, provide an interesting counterpoint to shared UI systems (e.g., *JCE*)—they provide the same *Complete* automation and compiled code reuse of *Any* programmer-defined semantic object. However, they use the repository rather the UI as a shared medium. Hence, for performance reasons, the sharing is effectively limited to asynchronous mode, and no sharing of UI objects is feasible. Furthermore, having only access to the persistent state of the application, *DFS* has no idea of the user interface.

In Chapter 5, we showed how we can simulate the basic DFS asynchronous sharing model, which is based on users taking turns at updating a shared object. To turn this into a true file system simulation, we can define a generic *File* object as a sequence of bytes and the directories as sequences of *File*s. Thus, files would appear as nodes a tree structure and updates to different files would be detected as non-conflicting (as in a regular DFS).

6.3.9 Coda

In the evaluation of *Coda*, we represent two separate evaluation points, which correspond to the two ways in which the infrastructure can be used. The first one is its basic operation, which is practically identical to traditional DFS. The main difference here is the option of trickle integration, which is primarily a performance consideration and has virtually no bearing on the sharing semantics. The second option for using *Coda* is to take advantage of the opportunity to provide procedures for automatic conflict resolution. This considerably improves the sharing flexibility of the application but comes at a high development cost. Since conflict resolutions procedures are not needed until an actual conflict is flagged, the infrastructure can support late binding of these routines, which serve as procedural specification of the sharing mode. As already noted, there is little in the way of automation in this process and using procedural sharing specifications (in the form of conflict resolution routines) make user-level specification difficult. The last difference is the separation of the shared object (file) and its sharing implementation—the conflict-resolution procedure.

The simulation of *Coda* is similar to that of traditional DFS. The type-specific procedures supplied by the programmer can be implemented as *merge* handlers in our infrastructure.

That is, we can take the conflict resolution procedure and place it as a property handler in the property specification. At run time, the infrastructure will lookup and invoke the handler to update a replica.

6.3.10 Lotus Notes

Notes presents a challenge in terms of evaluation because of its peculiar sharing model. Unlike the rest of the surveyed systems, the notions of replication and replica consistency are entirely decoupled. The replication of a document is controlled by a set of replication attributes that determine with whom a *Notes* document is shared. However, the system makes no attempt to reconcile concurrent updates to a document—it merely flags the conflict, creates a new version of the document and leaves the users to sort it out. In other words, conflict detection is coarse-grained and automatic, whereas conflict resolution is manual. Thus, in terms of our requirements, *Notes*' evaluation is similar to that of DFS, with the exception of abstraction and specification flexibility. Unlike DFS, where the shared object is opaque to the system, *Notes'* sharing model is based on a database of document records. Thus, the system is able to provide a variety of user-controlled replication and access control parameters, which in terms of out requirements means late specification binding and *High* ease of specification.

To simulate *Notes* sharing model, we define represent its documents is as objects with a sequence of document versions—one primary and a number of *response* ones. For example,

```
public class NotesDocument implements Identifiable {
  public GID getGID();
  public void setGID(GID gid);
  public void addVersion(NotesVersion version);
  public void removeVersion(NotesVersion version);
  …
}
```

Whenever a local replica receives a remote document with the same identifier, it does not attempt to merge but rather adds it to the list of version. In this case, the *GID* property provides a convenient unique identifier, which can be used as a key into a *Notes* database. The database itself can readily be represented by a table, and concurrent updates can be detected using the document's GID. Since conflicts are detected late and "handled" in a predefined manner, we need a simple *merge* handler for the database table, which adds a new version should it detect a conflict with the local database.

Simulating the replication model requires some more work, but can still be achieved within the framework of our model. Recall that we can specify the sharing for any subgroup of users and, in particular specify whether the entity should be shared at all. Hence, what is needed is to store persistently those parameters so that whenever changes are committed they would be communicated to the respective remote users.

One aspect we cannot accommodate is the replication architecture (e.g., peer-to-peer, hub-and-spoke, tree) because these are parameters of the communication layer that we do not control. Another aspect is access control, which is an issue we have not so far addressed and is one of the first problems we plan to address in the future.

6.3.11 Bayou

The main feature of *Bayou* in terms of fulfilling our requirements is its use of conflict-detection queries and procedures. This allows its mechanisms to be composed with any application because the translation between shared application structures and *Bayou*'s tuples is entirely the responsibility of the developer. This results in increased flexibility, satisfying our semantic object sharing and late specification requirements, but also implies reduced automation reflected on our *Low* scores on automation and ease of specification. Furthermore, as with all asynchronous sharing system, the UI cannot be shared directly.

Code reuse was never on *Bayou*'s agenda so programmers must modify the single-user application to use the system's mechanisms. Support for extensibility is a byproduct of the fact that developers must write an intermediate layer of adaptation code that allows the application to use infrastructure services. Hence, if the intermediate code is structured properly, the application and the infrastructure implementation can be varied independently.

Like other database-oriented infrastructures, *Bayou*'s tuples can be represented as objects consisting of simple properties that are being inserted into and removed from a table. User can write specialized merge handlers for the tables that can perform queries and automatically resolve conflicts. In fact, the query and conflict resolution parts can be separated into different handlers with the latter being invoked only when a conflict arises.

The one feature we do not directly support is the query language. However, it could be developed as a separate infrastructure service with potentially more sophisticated features. In particular our knowledge of the structural dependencies within the shared objects would

allow us to detect conflicts at a finer granularity by using a version of the object diff-ing routine. In this case, we would not keep a shadow copy of the objects but would compare two objects being inserted under the same key to find out if this is a true conflict by examining their differences at a finer granularity using property-based decomposition.

6.3.12 TACT

TACT presents an interesting point of evaluation because its goals and implementation are, to a large extent, complimentary to our own. It exclusively focuses on the problem of implementing a flexible consistency model that is independent of the application's shared object structures. Rather, it is up to the application programmer to perform the non-trivial task of mapping those structures to the infrastructure-defined abstraction of a *conit* (similar to a database record). Once the mapping is completed, the infrastructure automatically provides the sharing of the structures. Hence, overall, this scheme provides a *Moderate* level of automation.

One major problem with *TACT*'s read/write database model is that it is too restrictive and does not perform well for dynamic structures that grow/shrink as a result of user actions. As already discussed in the context of *JavaBeans'* model, the main issues are the detection of false conflicts and the inability to reconcile concurrent user updates. Thus, overall, we evaluate the sharing flexibility as *Moderate.*

Since the developer is tasked with translating shared objects into *conits*, there are few limitations on abstraction flexibility (albeit at the expense of automation). The sharing specification is based on parameters so it supports late binding. A parameter-based mechanism is generally easy to specify, however, in this case the specification metrics are somewhat unusual and it is not always obvious how they map to changes in the sharing behavior,. Therefore, we consider it a system with *Moderate* ease of specification.

TACT implicitly requires that the application define conits and explicitly manage their dependencies. Thus, it is not possible to directly reuse compiled code but it conceivable to make the application gradually collaboration-aware step by step. The system is not designed for extensibility, which is not surprising given that it attempts to subsume most consistency models currently in use.

Overall, our infrastructure can simulate *TACT* by defining read/write handlers that implement its model and by assuming (as it is done in the original implementation) that applications only preannounce the updates and these are carried out by the infrastructure at the appropriate time. However, this approach would be tantamount to re-implementing *TACT* and, therefore, we are interested in interfacing our infrastructure with *TACT* so that we can get the benefits of its generic consistency model at a low specification cost. In particular, we believe that properties are good a basis for defining *conit*s and structural dependencies can automate the process of identifying dependencies between read and write operations.

6.3.13 Sync

Due to its predefined shared abstractions, which can be directly employed by the application developer, *Sync* provides a *High* level of automation. Once the shared structures are built from the shared primitives, the infrastructure handles the sharing process automatically. The system allows programmers/users to specify a wide range of sharing (conflict resolution) policies. Originally, the system was limited to synchronous sharing, however, follow up work has demonstrates its use for synchronous sharing. Supporting intermediate, *Suite*-like, sharing modes cannot be currently accommodated.

Sync provides sharing of semantic objects that either directly use the shared classes, or inherit from them. Hence, the sharing is independent from the user interface aspect of the application—in fact, *Sync* assumes that the semantic object and its UI is implemented in separate classes and the UI classes are ignored for the sharing purposes, thus the UI cannot be shared

Sync was designed to provide flexible table-driven merging, which implies late, run-time binding of policy and mechanism. Furthermore, it provides multiple ways of specifying default policies, which considerable lowers the specification effort on part of the user.

Code reuse was not a primary goal directly addressed by *Sync*. As our discussion in Chapter 2 showed, converting a single-user application into a multi-user one requires re-engineering of its class hierarchy, which implies that compiled code reuse is not supported. However, the infrastructure does support a gradual introduction of collaboration awareness in that the programmer may start with some default policies, then move on to custom policies

specified by merge tables, and eventually define additional replicated classes should the standard ones prove inadequate.

Finally, *Sync*'s design does not address our extensibility requirements the way other *Java*-based infrastructures—such as *DISCIPLE* and *JViews*—do. Although by virtue of implicitly using *Java*'s dynamic class loading, it is technically possible to replace the implementation of the replicated classes without modifying the application, we do not consider *Sync* to fulfill our requirement of separating the shared abstraction and sharing mechanism because of the use of inheritance, which binds the abstraction and mechanism together. Along the same lines, the infrastructure was not designed to accommodate new functions by simple composition.

A simulation of *Sync*'s merge model fits well in our infrastructure model. The hierarchical top-down approach is analogous to what we used in our basic design for infrastructure services, such as diff-ing. *Sync*'s `ReplicatedRecords` correspond to an object being decomposed into a set of properties. *Sync*'s replicated basic types—int, string, etc—map to simple read/write properties in our model, whereas its complex types—`ReplicatedSequence` and `ReplicatedDictionary`—correspond to objects with a single *sequence/table* property, respectively, and would be covered by the standard property specifications that are part of our infrastructure. Hence, only suitable *merge* handlers that deal with merging of individual properties (based on merge tables) must be defined to complete the simulation of a *SyncClient*.

To simulate the *SyncServer* part of the system, we need a master copy version of our architecture like ones we used in the examples in Section 5.1. Accordingly, a server version of the merge procedure must also be developed. The main difference here is that the server merge procedure may have to respond back with a list of property operations that a client must execute to achieve consistency. Naturally, a commit-based sharing policy would be used to control the sharing on both ends.

6.3.14 Our Infrastructure

Column 'Our' of Table 6.1 (repeated in Table 6.2) summarizes the evaluation of our infrastructure with respect to the original requirements. The assessment is made based on the

detailed discussion in Chapters 3, 4, and 5, and the following points provide a recapitulation of the essential arguments behind our claims with respect to each requirement.

- *Automation*. Our support for automation falls into three separate cases, following our basic classification of sharing scenarios in Chapter 1. For user interface-based sharing, we provide WYSIWIS sharing equivalent to that of other systems at no development cost. For application-based sharing, we provide a functionally equivalent mechanism to *Colab*'s broadcast method although without the compiler support. We provide direct automatic sharing of the abstractions supported by *GroupKit*, *Suite*, and *Sync*. For repository-based infrastructures, such support can be added at the nominal cost of providing property specifications and appropriate handlers for each property type. For infrastructures that rely on conflict resolution, such as *Coda* and *Bayou*, this translates into providing *merge* handlers, in addition to the required *read* handler.
- *Sharing Flexibility.*
  - *Concurrent updates.* Our infrastructure, like most of the rest, supports concurrent updates from multiple users. The only exception is UI-based sharing where simultaneous access must be blocked to ensure correctness.
  - *Semantic sharing.* As our discussion in Chapter 5 showed, our work can support the flexible sharing of semantic objects independent of their user interface
  - *User interface sharing.* Similarly, we demonstrated sharing that can be applied to the user interface of any semantic object and, thereby, implement UI-based sharing of the abstraction.
  - *Sharing modes.* Our sharing model supports the sharing provided by *Suite*, which is the infrastructure with the highest sharing flexibility for application-based sharing. Also, by explicitly separating the event processing into three distinct phases and adding parameters to control each phase, our extension of the model allows a finer grain control of the sharing process and can integrate as special cases a wider variety of sharing mechanisms in use today. By introducing a description mechanism for application layers, we enabled higher-level control of the application sharing that in addition to accommodating the layer-based sharing of existing systems, allows multi-layer sharing to be implemented under more general assumptions than currently possible.

- *Abstraction Flexibility*
  - *Programmer-defined semantic objects.* Again, we have two cases. The first one is the use of *Colab*-like broadcast methods, which can be correctly applied to (almost) any object under the assumption of synchronous sharing. The second one the sharing of programming pattern-based objects. As already discussed, all of the abstractions shared by other infrastructures can be modeled as special cases of the use of patterns. At the same time, the class of pattern-based objects that can automatically be shared is a proper superset of the ones supported by any of the other systems.
  - *Programmer-defined user interface.* As with most other sharing infrastructures, our sharing mechanisms are independent of the user interface implementation. Furthermore, the described experience in developing applications shows that they can be applied to any application layer. Notably, by using the programmer-specified dependencies, we can enable the simultaneous sharing of multiple layers that do not depend on each other, as well as prevent the simultaneous sharing of dependent layer to ensure correctness.
- *Specification flexibility.*
  - *Late specification binding.* Our parameter-based specifications have their own external representation based on *XML*. Therefore, they are completely independent of the sharing mechanisms that implement them and the two are composed at run time. Moreover, even the binding of procedural specifications, such as a conflict detection routine, is performed at run time through a property specification.
  - *Ease of specification.* Our specification model builds on that of *Suite*'s and, therefore, our infrastructure provides a level of support that is at least as high as that of *Suite.* Our main task has been to adapt *Suite*'s inheritance-based specification approach so that it can be applied to object-based programming. Going a step further, we have provided the policy specifications with a standardized external representation. This permits an administrator to assume the responsibility of defining a set of standard policies, which end-users can employ without knowing the details of the sharing model.

- *Code reuse*
  - *Compiled code reuse.* Code reuse has been a major objective of our work and we claim that we have satisfied this goal better than existing infrastructures. Specifically, for UI- and repository-based sharing, our infrastructure can provide complete automation, such as the one exhibited by other infrastructures. However, we also address reuse in the context of application-based sharing, which, as our evaluation table shows, has been achieved by very few systems. In particular, out of the surveyed systems, only *JViews* and *DISCIPLE* have such mechanisms and we showed that both can be handled as special cases of our pattern-based approach. Recall that the set of object properties and the corresponding property handlers can be extended without modifying the shared objects. Hence, we can potentially add support for arbitrary new types of shared objects without recompiling existing code, which is not possible with any of the other systems.

    Another aspect of our support for reuse is the ability to reuse handler code across objects that belong to unrelated branches of the class hierarchy. In our model description we showed that patterns are a more general concept than interfaces and, therefore, an implementation based on patterns is more general and can be directly reused with a wider range of objects.
  - *Incremental collaboration awareness.* As we showed in Chapter 5, our system facilitates a development approach in which collaboration features are incrementally introduced into the application. At each step, a new set of default capabilities become available, enabling the programmer to minimize the development effort. For example, if users need only asynchronous sharing, there is no need for the application to implement a notification mechanism. Furthermore, if certain services are not directly needed, the programmer need not provide them. For example, if *diff*-ing is not needed, the application does not need to specify any *diff* handlers.
- *Extensibility.* Our work supports all three aspects of this requirement—separation of shared abstraction and implementation, separation of infrastructure functions, and late component binding. The main mechanism implementing this support is, again, the property specification language, which allows the composition of shared abstractions and

sharing functions to be performed late. Similarly, different infrastructure functions may interact indirectly, using the specification as a means of discovering each other.

7. CONCLUSIONS AND FUTURE WORK

A common approach employed by sharing infrastructures is to present the programmer with shared programming abstractions as a basis for implementing multi-user applications. We have identified a set of generic automation, reuse and flexibility requirements that such sharing infrastructures should satisfy. However, our survey of a number of influential systems lead us to the conclusion that, while individual requirements have been met, none of the existing infrastructures fulfills a substantial subset of the requirements as a whole. Therefore, we have developed a new model and a corresponding prototype implementation of an infrastructure for sharing of distributed objects that fulfills the set of requirements better than current systems. Below we present our conclusions, as well as ideas on extending this research in the future.

7.1 Conclusions

Based on the presented results of our work, we draw the following main conclusions:

- For each of the generic infrastructure requirements identified in Chapter 1—automation, flexibility, code reuse, and extensibility—there exists at least one system that satisfies it to high degree. However, no single system fulfills all of the requirements, or even a substantial subset of them.
- Our new object-sharing infrastructure better satisfies the infrastructure requirements than current systems. That is, for any given sharing scenario, it satisfies the requirements at least as well as existing systems and, for a large number of scenarios, it satisfies a subset of the requirements to a higher degree than existing systems. The developed conceptual model and prototype implementation make several research contributions:
    - The introduced property specification language formalizes the notion of a programming pattern and extends the abstraction flexibility of the sharing infrastructure. We showed that the shared abstractions supported by other systems can be modeled as special cases of the use of patterns.

- The developed component architecture, based on property handlers, provides a means to completely separate the core application code from the infrastructure services by providing a means to flexibly compose them based on specifications.
- The introduced application layer specification enables the generic description of application layers and their dependencies, thereby enabling the dynamic switching of the shared layer at run time, while maintain correct sharing semantics.
- The introduced sharing model, based on the original *Suite* coupling model, provides a more general framework for controlling the sharing semantics than existing approaches. This includes the incorporation of new services, such as object diff-ing, and the integration of a number of additional sharing options that allow the system to support the sharing models of other infrastructures.

- The developed formal pattern specification language and the associated component architecture based on property handlers are generic software engineering tools. We have demonstrated their use outside the immediate scope of collaboration and we have concrete ideas to apply them to a number of other areas, which we discuss below as part of our future work.
- Our infrastructure model does not explicitly incorporate a number of collaboration functions, such as access control and multi-user undo/redo, and support for others, such as concurrency control and session management is limited to the minimum required to demonstrate a working sharing implementation. While our design was developed with such extensions in mind, clearly, further work is needed to achieve a comprehensive sharing infrastructure model. In the following section we discuss our ideas in that respect.

7.2 Future Work

Our ideas on extending this work can be grouped into three distinct categories that are discussed as follows. First, we consider the extension of the current infrastructure model and implementation to (ultimately) provide a comprehensive collaboration model that covers all aspects of the design space for collaborative applications. Such an extension is to be accomplished in two ways: adding new collaboration services and integrating our work with existing systems that are complementary. Next, we present our thoughts on improving our

pattern-based mechanisms to make them more general and easier to use. Finally, we give our ideas on applying pattern-based approaches to non-collaborative applications.

7.2.1 Collaborative Infrastructure Extensions

7.2.1.1 Service Extensions

Recall that we have only explicitly addressed concurrency control only in the limited case of UI-based sharing (Section 5.2) for purposes of correctness. By concurrency control we mean a mechanism that allows a group of collaborators to explicitly coordinate their actions to ensure the desired level of consistency of the shared artifact. There is a wide range of useful concurrency control schemes employed by existing systems. On the one end, we have the most liberal option of no application-mediated control (implicitly assumed by most of our current work), which relies on users coordinating their actions through social protocol. On the other end, we have the most conservative scheme, which only allows one user at a time to modify the shared object (as in UI-based or file-based sharing). In between, there are numerous other options that can broadly be classified into optimistic and pessimistic. The former seek to maximize concurrency by assuming that conflicts are rare and, therefore, in most cases it is advantageous to allow concurrent user actions to proceed and to defer conflict detection/resolution to a later time. The latter places an emphasis on providing firm consistency guarantees, often at the price of lower concurrency, by using enforcement mechanisms that prevent inconsistencies altogether A pessimistic approach inherently implies that the infrastructure must have complete control over updates of the shared object. Hence, we would need a mechanism by which the application (*pre-*)*announces* intended changes and the infrastructure has the power to prevent them from taking place. Accordingly, we would need to extend our event model to accommodate a two-phase announce/notify protocol.

Our ultimate goal with respect to concurrency control would be to develop a model that accommodates as many of the currently existing options as possible under one roof, and to allow users to dynamically select the scheme that fits their collaboration best. Furthermore, we would like to provide multiple levels of support based on the notification mechanism implemented by the application. In particular, since most applications do not employ two-phase notification, the open issue we seek to explore is: how much support can we provide

under these circumstances? Is it possible to automatically generate a pre-announcement on behalf of the application or compensate for the lack of pre-announcement through an undo mechanism? Another interesting research topic is to explore the opportunities for inter-operation among different concurrency control schemes that fit the developed model.

Access control determines which users have access to various operations on a protected object. By definition, access control requires a preventive enforcement mechanism that has the power to effectively deny access to the object's operation by an unauthorized user. Our goal would be to develop an access control model that can be composed with the rest of our infrastructure.

Our infrastructure would also benefit from the presence of a generic multi-user undo/redo mechanism similar to that of *Suite*. In addition to resolving the immediate issues of the semantics of the undo command, it could also help in implementing some of the concurrency control options outlined above. In particular, an interesting issue is, given a generic undo/redo mechanism, which requires only update notification (and not pre-announcement), is it possible to effectively simulate pessimistic concurrency control without requiring pre-announcement. That is, implement a scheme under which it may be possible for an object replica to temporarily become inconsistent (due to an unauthorized local operation being applied first notified later) but be automatically brought back into consistency through a corresponding undo operation.

7.2.1.2 Integration with Other Infrastructures

To a certain degree, the concrete implementation of our infrastructure has been influenced by the idea of integrating certain functions already implemented by other systems with our own. A prime example in that respect is session management. As our description in Chapter 4 shows, we have implemented the bare minimum necessary to test our infrastructure. Our rationale has always been to eventually utilize the comprehensive session management mechanism offered by *JCE*. Such an integration would also strengthen our argument that we have developed an extensible infrastructure.

Another complimentary mechanism we would like integrate with is *Sync*'s table-driven merge mechanism. Recall that our model allows for custom *merge* property handlers to be defined, thereby enabling the simulation of *Sync*'c merging model. However, we have chosen

not to replicate *Sync*'s full implementation effort largely because we believe that our work can be composed with *Sync*'s merging mechanism with a reasonable effort. Furthermore, we would like to explore the integration with our object diff-ing mechanism, which is a natural complement to merging.

Another direction for our integration work would be an effort to compose our work with *Chung*'s generic logging mechanism, which provides for *replication flexibility*—the ability to dynamically change the physical replication architecture of the application to adapt to network conditions and user preferences. In particular, we believe that we can leverage knowledge of the object and application structures (through properties and layers, respectively), as well as the *semantics* of property events, to automate the process of mapping application-specific communication protocols to the generic I/O protocol upon which *Chung*'s work is based.

7.2.2 Pattern Specification Mechanism Improvements

One of the practical issues we have encountered in our work with property specification has been the need to verify that the *XML*-based specifications do indeed find the properties they are designed to find. A wrong specification may lead to some properties not being recognized, which may be difficult problem to spot in a larger software project. Therefore we would like to develop more user-friendly tools that facilitate the specification process. As a first step, we would like to develop a specification-by-example tool that would allow users to bring up the list of methods for a particular classes and provide examples of specific patterns, based on which our infrastructure would deduce the exact pattern specification. A more challenging issue is to develop a tool that automatically examines the code of an application and generates a list of candidate property definitions to the developer.

7.2.3 Pattern-based Approaches to Non-Collaborative Applications

7.2.3.1 Automated Object Testing

As an ongoing project, we are currently developing a framework for automated object testing based on patterns. The fundamental idea behind it is to use the object property analysis to break down the problem of testing the whole object into a problem of testing a set of object properties. Each individual property comes with its own autonomous semantics that

could, in many cases, be tested separately. The testing of an individual property is based on the use of *Guttag*-style algebraic specifications. We use the *semantics* attribute classification of pattern methods (*constructor*, *destructor*, *modifier*, and *accessor*) to automatically generate test cases. The basic idea is to generate a large set of test cases by performing all possible combinations of *constructor* method invocations using the test points as arguments. For example, if an object has a property with constructor method called `addElement`, and a set of three test points—`"1"`, `"2"`, and `"3"`—the test cases would be generated as follows:

```
object.addElement("1")
object.addElement("1"). addElement("1")
object.addElement("1"). addElement("1"). addElement("1")
object.addElement("1"). addElement("1"). addElement("2")
object.addElement("1"). addElement("1"). addElement("3")
object.addElement("1"). addElement("2"). addElement("1")
...
object.addElement("2")
object.addElement("2"). addElement("1")
…
object.addElement("3")
...
object.addElement("3").addElement("3").addElement("3")
```

For each generated test case, the infrastructure invokes each of the other methods and compares the result with the expected one by invoking a corresponding programmer-defined predicate. For example, for a `removeElement` *destructor* method, the developer must specify the following predicate:

```
boolean removeElementTest( oldState, newState, arguments)
```

The predicate invocation can be interpreted as asking folloing question:

Given the `oldState` of the object, the fact that the `removeElement` method has been invoked with the given `arguments`, does the observed `newState` agree with the desirable results?

Ideally, the above question should be answerable based on formal axiomatic specifications, however, even using this procedural form, which is compatible with the de facto standard *JUnit*[18] testing tool we expect to see considerable benefits. Namely, with a comparable effort development effort, our infrastructure will perform automatic exhaustive

[18] `http://www.junit.org`

testing of all test point combinations as opposed to the manual ad-hoc testing practices that are prevalent today.

To address the fact that properties may have dependencies, we apply a similar exhaustive testing idea to test combinations of properties. While this results in a combinatorial that may become too computationally expensive, it is quite possible to control it by seeking information from the developer on which properties are, in fact, independent and exclude them from the test case generation. Conversely, the testing can help in certifying that the properties are indeed independent as expected by the developer.

Finally, we would like to take the pattern a step further to allow automated testing based on *design patterns*. At the very minimum, we currently have the basic mechanisms to automatically match objects that implement recognized design patterns by examining their programming patterns. For example, one could define *observer* and *observable* programming patterns and recognize that an object implementing the former and an object implementing the latter may, together, be implementing a design pattern.

7.2.3.2 Structured Code Generation

Following a set of programming patterns is especially desirable in any software engineering team because members interact with each other's code. In our work we have primarily used property specifications to identify patterns in already existing code. However, it is quite possible to perform the reverse process, in which case the users can provide values for the free variables in the pattern specifications, and allow the infrastructure to automatically generate the application code, which would be guaranteed to follow the given patterns.

8. APPENDIX

8.1 XML Property Specification Schema

```
<?xml encoding="UTF-8"?>
<!ENTITY % EXPR "reference | plural | short | plural_short">
<!ENTITY % PATTERN_REF "reference">

<!ENTITY VOID   "<literal>void</literal>">
<!ENTITY INT    "<literal>int</literal>">
<!ENTITY BOOL   "<literal>boolean</literal>">
<!ENTITY STRING "<literal>java.lang.String</literal>">
<!ENTITY ENUM   "<literal>java.util.Enumeration</literal>">
<!ENTITY ARRAY  "<literal>[]</literal>">
<!ENTITY VECTOR "<literal>java.util.Vector</literal>">

<!ELEMENT property_spec (pattern+, constraint*, name_rule, type_rule,
                         exclude*, handler+)>
<!ATTLIST property_spec
          type CDATA #REQUIRED
          version CDATA #REQUIRED>
<!ELEMENT pattern (return_type, method_name, argument_type*)>
<!ATTLIST pattern name ID #REQUIRED>
<!ATTLIST pattern semantics (unknown | accessor | constructor |
                             modifier | destructor) "unknown">
<!ELEMENT return_type (literal | variable | (variable,literal) |IGNORE)>
<!ELEMENT argument_type (literal | variable | IGNORE)>
<!ELEMENT method_name (literal | (literal?, variable, literal?)) >

<!ELEMENT literal (#PCDATA)>
<!ELEMENT variable EMPTY>
<!ATTLIST variable name ID #REQUIRED>
<!ELEMENT IGNORE EMPTY>

<!ELEMENT constraint (lhs, rhs)>
<!ATTLIST constraint predicate (equals | subclassof) "equals">
<!ELEMENT lhs ((%PATTERN_REF;), (%EXPR;))>
<!ELEMENT rhs ((%PATTERN_REF;), (%EXPR;))>

<!ELEMENT reference EMPTY>
<!ATTLIST reference variable IDREF #REQUIRED>

<!ELEMENT short EMPTY>
<!ATTLIST short variable IDREF #REQUIRED>

<!ELEMENT plural EMPTY>
<!ATTLIST plural variable IDREF #REQUIRED>

<!ELEMENT plural_short EMPTY>
<!ATTLIST plural_short variable IDREF #REQUIRED>

<!ELEMENT name_rule (literal | (%EXPR;))>
<!ELEMENT type_rule (literal | reference)>

<!ELEMENT exclude EMPTY>
```

```
<!ATTLIST exclude
          class_name CDATA #REQUIRED
          name CDATA #REQUIRED
          scope (subclass | self} "subclass">

<!ELEMENT handler EMPTY>
<!ATTLIST handler
          operation CDATA #REQUIRED
          class_name CDATA #REQUIRED>
```

8.2 Generic TABLE Property Specification

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE property_spec SYSTEM "PropertySpec.dtd">

<!-- "table" property version 1.1
  Example (from java.util.Hashtable):
  public Object put(Object obj, Object value)
  public Object remove(Object obj)
  public Object get(Object key)
  public Enumeration keys()

  name: Element
  type: Object
-->

<property_spec type = "table" version = "1.1">
  <!-- "bind" pattern -->
  <pattern name = "bind">
    <return_type><IGNORE/></return_type>
    <method_name>
      <literal>put</literal>
    </method_name>
    <argument_type>
      <variable name = "BindKeyType"/>
    </argument_type>
    <argument_type>
      <variable name = "BindValueType"/>
    </argument_type>
  </pattern>
  <!-- "unbind" pattern -->
  <pattern name = "unbind">
    <return_type><IGNORE/></return_type>
    <method_name>
      <literal>remove</literal>
    </method_name>
    <argument_type>
      <variable name = "UnbindKeyType"/>
    </argument_type>
  </pattern>
  <!-- "lookup" pattern -->
  <pattern name = "lookup">
    <return_type>
      <variable name = "LookupValueType"/>
    </return_type>
    <method_name>
      <literal>get</literal>
```

```
    </method_name>
    <argument_type>
      <variable name = "LookupKeyType"/>
    </argument_type>
  </pattern>
  <!-- "get_keys" pattern -->
  <pattern name = "get_keys">
    <return_type>
      &ENUM;
    </return_type>
    <method_name>
      <literal>keys</literal>
    </method_name>
  </pattern>
  <!-- BindKeyType == UnbindKeyType -->
  <constraint predicate = "equals">
    <lhs>
      <reference variable = "bind"/>
      <reference variable = "BindKeyType"/>
    </lhs>
    <rhs>
      <reference variable = "unbind"/>
      <reference variable = "UnbindKeyType"/>
    </rhs>
  </constraint>
  <!-- BindKeyType == LookupKeyType -->
  <constraint predicate = "equals">
    <lhs>
      <reference variable = "bind"/>
      <reference variable = "BindKeyType"/>
    </lhs>
    <rhs>
      <reference variable = "lookup"/>
      <reference variable = "LookupKeyType"/>
    </rhs>
  </constraint>
  <!-- BindValueType == LookupValueType -->
  <constraint predicate = "equals">
    <lhs>
      <reference variable = "bind"/>
      <reference variable = "BindValueType"/>
    </lhs>
    <rhs>
      <reference variable = "lookup"/>
      <reference variable = "LookupValueType"/>
    </rhs>
  </constraint>
  <name_rule>
    <literal>Table</literal>
  </name_rule>
  <!-- typing rule -->
  <type_rule>
    &VOID;
  </type_rule>
  <!-- handlers -->
</property_spec>
```

8.3 Generic SEQUENCE Property Specification

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE property_spec SYSTEM "PropertySpec.dtd">

   <!-- "sequence" property version 1.1
   Example (from java.util.Vector):
     public void insertElementAt(Object obj, int index)
     public void removeElementAt(int index)
     public void setElementAt(Object obj, int index)
     public Object elementAt(int index)
     public int size()

     name: Element
     type: Object
   -->
   <property_spec type = "sequence" version = "1.1">
     <!-- "insert" pattern -->
     <pattern name = "insert">
       <return_type><IGNORE/></return_type>
       <method_name>
         <literal>insert</literal>
         <variable name = "InsertName"/>
         <literal>at</literal>
       </method_name>
       <argument_type>
         <variable name = "InsertType"/>
       </argument_type>
       <argument_type>&INT;</argument_type>
     </pattern>
     <!-- "remove" pattern -->
     <pattern name = "remove">
       <return_type><IGNORE/></return_type>
       <method_name>
         <literal>remove</literal>
         <variable name = "RemoveName"/>
         <literal>at</literal>
       </method_name>
       <argument_type>&INT;</argument_type>
     </pattern>
     <!-- "set" pattern -->
     <pattern name = "set">
       <return_type><IGNORE/></return_type>
       <method_name>
         <literal>set</literal>
         <variable name = "SetName"/>
         <literal>at</literal>
       </method_name>
       <argument_type>
         <variable name = "SetType"/>
       </argument_type>
       <argument_type>&INT;</argument_type>
     </pattern>
     <!-- "count" pattern -->
```

```
<pattern name = "count">
  <return_type>&INT;</return_type>
  <method_name>
    <literal>size</literal>
  </method_name>
</pattern>
<!-- "lookup" pattern -->
<pattern name = "lookup">
  <return_type>
    <variable name = "LookupType"/>
  </return_type>
  <method_name>
    <variable name = "LookupName"/>
    <literal>at</literal>
  </method_name>
  <argument_type>&INT;</argument_type>
</pattern>
<!-- InsertType == SetType -->
<constraint predicate = "equals">
  <lhs>
    <reference variable = "insert"/>
    <reference variable = "InsertType"/>
  </lhs>
  <rhs>
    <reference variable = "set"/>
    <reference variable = "SetType"/>
  </rhs>
</constraint>
<!-- SetType == LookupType -->
<constraint predicate = "equals">
  <lhs>
    <reference variable = "set"/>
    <reference variable = "SetType"/>
  </lhs>
  <rhs>
    <reference variable = "lookup"/>
    <reference variable = "LookupType"/>
  </rhs>
</constraint>
<!-- InsertName == RemoveName -->
<constraint predicate = "equals">
  <lhs>
    <reference variable = "insert"/>
    <reference variable = "InsertName"/>
  </lhs>
  <rhs>
    <reference variable = "remove"/>
    <reference variable = "RemoveName"/>
  </rhs>
</constraint>
<!-- RemoveName == SetName -->
<constraint predicate = "equals">
  <lhs>
    <reference variable = "remove"/>
    <reference variable = "RemoveName"/>
  </lhs>
  <rhs>
```

```
        <reference variable = "set"/>
        <reference variable = "SetName"/>
      </rhs>
    </constraint>
    <!-- SetName == LookupName -->
    <constraint predicate = "equals">
      <lhs>
        <reference variable = "set"/>
        <reference variable = "SetName"/>
      </lhs>
      <rhs>
        <reference variable = "lookup"/>
        <reference variable = "LookupName"/>
      </rhs>
    </constraint>
    <!-- naming rule -->
    <name_rule>
      <reference variable = "InsertName"/>
    </name_rule>
    <!-- typing rule -->
    <type_rule>
      <reference variable = "InsertType"/>
    </type_rule>
    <!-- handlers -->
  </property_spec>
```

8.4 SEQUENCE Property Specification for java.awt.Component

```
<property_spec type = "sequence" version = "2.1">
   <!-- Example: java.awt.Component
     add:    public java.awt.Component add(java.awt.Component comp);
     remove: public void remove(java.awt.Component comp);
     list:   public java.awt.Component[] getComponents();

   -->
  <!-- "add" pattern -->
  <pattern name = "add">
    <return_type>
      <IGNORE/>
    </return_type>
    <method_name>
      <literal>add</literal>
    </method_name>
    <argument_type>
      <variable name = "AddType"/>
    </argument_type>
  </pattern>
  <!-- "remove" pattern -->
  <pattern name = "remove">
    <return_type>
      <IGNORE/>
    </return_type>
    <method_name>
      <literal>remove</literal>
```

```
  </method_name>
  <argument_type>
    <variable name = "RemoveType"/>
  </argument_type>
</pattern>

<!-- "list" pattern -->
<pattern name = "list">
  <return_type>
    <variable name = "ListType">
    &ARRAY;
  </return_type>
  <method_name>
    <literal>get</literal>
    <variable name = "ListName"/>
  </method_name>
</pattern>

<!-- AddType == RemoveType -->
<constraint predicate = "equals">
  <lhs>
    <reference variable = "add"/>
    <reference variable = "AddType"/>
  </lhs>
  <rhs>
    <reference variable = "remove"/>
    <reference variable = "RemoveType"/>
  </rhs>
</constraint>

<!-- AddType == ListType -->
<constraint predicate = "equals">
  <lhs>
    <reference variable = "add"/>
    <reference variable = "AddType"/>
  </lhs>
  <rhs>
    <reference variable = "remove"/>
    <reference variable = "ListType"/>
  </rhs>
</constraint>

<!-- plural(short(AddType)) == ListName -->
<constraint predicate = "equals">
  <lhs>
    <reference variable = "add"/>
    <plural_short variable = "AddType"/>
  </lhs>
  <rhs>
    <reference variable = "list"/>
    <reference variable = "ListName"/>
  </rhs>
</constraint>

<name_rule>
  <reference variable = "ListName"/>
</name_rule>
```

```
  <!-- typing rule -->
  <type_rule>
    <reference variable = "AddType"/>
  </type_rule>

  <!-- handlers -->
  <handler operation = "read" class_name =
                       "colab.bus.property.handlers.SequenceReader"/>
  <handler operation = "write" class_name =
                       "colab.bus.property.handlers.ArrayWriter"/>
</property_spec>
```

8.5 SET Property Specification for GraphDraw

```
<property_spec type = "set" version = "1.1">
  <!-- "list" pattern -->
  <pattern name = "list">
    <return_type>
      &VECTOR;
    </return_type>
    <method_name>
      <literal>get</literal>
      <variable name = "ListName"/>
    </method_name>
  </pattern>

  <!-- "add" pattern -->
  <pattern name = "add">
    <return_type>
      <IGNORE/>
    </return_type>
    <method_name>
      <literal>add</literal>
    </method_name>
    <argument_type>
      <variable name = "AddType"/>
    </argument_type>
  </pattern>

  <!-- "remove" pattern -->
  <pattern name = "remove">
    <return_type>
      <IGNORE/>
    </return_type>
    <method_name>
      <literal>remove</literal>
    </method_name>
    <argument_type>
      <variable name = "RemoveType"/>
    </argument_type>
  </pattern>

  <!-- AddType == RemoveType -->
  <constraint predicate = "equals">
    <lhs>
```

```
      <reference variable = "add"/>
      <reference variable = "AddType"/>
    </lhs>
    <rhs>
      <reference variable = "remove"/>
      <reference variable = "RemoveType"/>
    </rhs>
  </constraint>

  <!-- plural( short( AddType)) == ListName -->
  <constraint predicate = "equals">
    <lhs>
      <reference variable = "add"/>
      <plural_short variable = "AddType"/>
    </lhs>
    <rhs>
      <reference variable = "list"/>
      <reference variable = "ListName"/>
    </rhs>
  </constraint>

  <name_rule>
    <reference variable = "ListName"/>
  </name_rule>

  <!-- typing rule -->
  <type_rule>
    <reference variable = "AddType"/>
  </type_rule>

  <!-- handlers -->
  <handler operation = "read" class_name =
                       "colab.bus.property.handlers.VectorReader"/>
  <handler operation = "write" class_name =
                       "colab.bus.property.handlers.VectorWriter"/>
</property_spec>
```

9. REFERENCES

[1] Abdel-Wahab, H., Jeffay, K., *Issues, problems, and solutions in sharing clients on multiple displays.* Internetworking: Research and Experience, 1994. **5**: p. 1-15.

[2] Abdel-Wahab, H., Kvande, B., Kim, O. , Favreau, J.P. *An Internet Collaborative Environment for Sharing Java Applications*. in *5th Workshop on Future Trends of Distributed Computing Systems.* October 1997. Tunisia.

[3] Chabert, A., Grossman, E., *Java Object-Sharing in Habanero.* Communications of the ACM, June 1998. **41**(6): p. 69-76.

[4] Chung, G., Dewan, P., Rajaram, S. *Generic and Composable Latecomer Accommodation Service for Centralized Shared Systems.* in *HCI.* September 1998.

[5] Cohen, G.A., Jeffrey S. Chase, David L. Kaminsky. *Automatic Program Transformation with JOIE*. in *USENIX Annual Technical Symposium*. 1998.

[6] Coutaz, J., Nigay, L. *From Single-User Architectural Design to PAC*: A Generic Software Architecture Model for CSCW.* in *HCI.* 1997. Atlanta.

[7] Dewan, P., *Architectures for Collaborative Applications.* Trends in Software, special issue on Computer Supported Cooperative Work, 1998. **7**: p. 169-194.

[8] Dewan, P. and R. Choudhary, *A High-Level and Flexible Framework for Implementing Multiuser User Interfaces.* ACM Transactions on Information Systems, October 1992. **10**(4): p. 345-380.

[9] Dewan, P., Choudhary, R., *Coupling the User Interfaces of a Multiuser Program.* ACM Transactions on Computer Human Interaction, March 1995. **2**(1): p. 1-39.

[10] Dourish, P., Bellotti, V. *Awareness and Coordination in Shared Workspaces*. in *Proceedings of the ACM Conference on Computer Supported Cooperative Work*. 1992. Toronto, Ontario: ACM Press.

[11] Greenberg, S., Marwood, D. *Real time groupware as a distributed system: Concurrency control and its effect on the interface*. in *Proceedings of the ACM Conference on Computer Supported Cooperative Work*. 1994. Chapel Hill, North Carolina: ACM Press.

[12] Grundy, J. *Engineering component-based, user-configurable collaborative editing systems.* in *EHCI.* 1998.

[13] Hamilton, G., *JavaBeans specification*. 1997, Sun Microsystems.

[14] Hill, R.D. *The Abstraction-Link-View Paradigm: Using Constraints to connect User Interfaces to Applications*. in *In Human Factors in Computing Systems: CHI'92 Conference Proceedings*. 1992. Monterey, California.

[15] Krasner, G.E., Pope, S. T., *A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80.* Journal of Object-Oriented Programming, 1988. **1**(3): p. 26-49.

[16] L. Kawell Jr., S.B., T. Halvorsen, R. Ozzie, and I. Grief. *L. Kawell Jr., S. Beckhardt, T. Halvorsen, R. Ozzie, and I. Grief*. in *Proc. 2nd Cond. on Computer-supported Cooperative Work*. 1988.

[17] Mummert, L.B., Ebling, M.R., and Satyanarayanan, M., *Exploiting weak consistency for mobile file access.* Operating Systems Review, 1995. **29**(5): p. 143-155.

[18] Munson, J., Dewan, P., *Sync: a Java framework for mobile collaborative applications*, in *Computer*. 1997. p. 231-242.

[19] Richardson, T., Stafford-Fraser, Q. et al, *Virtual Network Computing.* IEEE Internet Computing, 1998. **2**(1).

[20] Roseman, M., Greenberg, S., *Building real-time groupware with GroupKit, a groupware toolkit.* ACM Transactions on Computer-Human Interaction, 1996. **3**(1): p. 66-106.

[21] Roussev, V., Dewan, P., Jain, V. *Composable Collaboration Infrastructures Based on Programming Patterns*. in *Proceedings of the 2000 ACM conference on Computer supported cooperative work*. 2000. Philadelphia, Pennsylvania.

[22] Stefik, M., et al., *Beyond the Chalkboard: Computer Support for Collaboration and Problem Solving in Meetings.* CACM, January 1987. **30**(1): p. 32-47.

[23] Tarpin-Bernard, F., David, B.T., Primet, P. *Frameworks and Patterns for Synchronous Groupware : AMF-C Approach*. in *EHCI.* September 1998. Greece.

[24] Terry, D.B., Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer and Carl H. Hauser. *Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System*. in *Proceedings of the fifteenth ACM symposium on Operating systems principles*. 1995. Copper Mountain, Colorado,.

[25] Yu, H., Vahdat, A. *Design and Evaluation of a Continuous Consistency Model for Replicated Services*. in *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI).* 2000.