A Model of Composite Objects for Information Mesh

by

Felix Tun-Han Lo

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Master of Engineering in Electrical Engineering and Computer Science

and

Bachelor of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 1996

Author . . . .
Department of Electrical Engineering and Computer Science
August 26, 1996

Certified by . . . .
Karen R. Sollins
Research Scientist
Thesis Supervisor

Accepted by . . . .
Morgenthaler
Chairman, Departmental Committee on Graduate Theses

A Model of Composite Objects for Information Mesh

by

Felix Tun-Han Lo

Submitted to the Department of Electrical Engineering and Computer Science
on August 26, 1996, in partial fulfillment of the
requirements for the degrees of
Master of Engineering in Electrical Engineering and Computer Science
and
Bachelor of Science in Electrical Engineering and Computer Science

Abstract

This report describes the design and implementation of a prototype for composite objects in the Information Mesh. A composite object is composed of a collection of objects. The prototype uses the example of a book. It was built using two recently developed technologies in distributed computing, Java and CORBA. Java supports mobile code systems, while CORBA provides the mechanism for making transparent requests. Both technologies are essential in building the prototype.

The prototype is then generalized into a composite model for all objects in the Information Mesh Object System. This general model extends from the current Mesh object model, and incorporates a new capability for the composition of objects. Objects of different types can be combined, while the mechanism for composition is transparent to users of the object system.

Thesis Supervisor: Karen R. Sollins
Title: Research Scientist

Acknowledgments

I would like to give my sincere thanks to:

Dr. Karen Sollins, my thesis supervisor, for her guidance throughout the project, and for the diligence with which she has read and understood the drafts of this thesis. Her ability to help me extract and clarify my ideas has been invaluable.

Lewis Girod, for helping me grasp many of the ideas in the Information Mesh.

I am also in debt to the following people, whose company has immensely enriched my life, both at MIT and beyond.

To Dad and Mom, Ada, Eileen, and Adrian, for their love and caring throughout these twenty-two years. No word can possibly express my gratitude toward my family.

To Andy, oyip, pchan, Anita, Jenny, Vinci, for sharing my walk at MIT since freshman year, and for being project partners in many classes.

To Jerome, Bernard, Yuk, Albert, the "old-biscuits", for helping me the way around MIT and making MIT a fun place to live in.

To khon, kennethp, for the happy time in UROP together and being two of my best friends.

To Cavlin, Chris, kclau and members in the hockey team, soccer team, and table-tennis team, for the wonderful memories together.

To David Ma, Jimmy, bwong, King, Mawlo, Kin, Richard, for walking together to find and grasp the right priorities in life.

To my friends in HKSS, for making these four years the happiest time of my life.

To the people in HKSBS, for their love and compassion through prayers and fellowship over these years.

Last but not least, to the Lord Jesus Christ, who has made all of the above a reality.

Contents

List of Figures

Chapter 1

Introduction

The Information Mesh is an infrastructure which provides better supports for distributed information applications. One of the major design goals of the Information Mesh is its evolvability. A flexible object-typing model has been developed so that the Information Mesh may evolve along with changing applications and supporting infrastructures [1]. Currently, the Information Mesh project has not developed a model of composite objects that takes advantage of the flexibility of the underlying object system. A composite object is composed of a collection of Mesh objects; it requires certain relationships among Mesh objects [3]. These relationship requirements reduce the flexibility allowed by the object system.

This work proposes a composite model which is compatible with the existing object system. The model is designed to retain the flexibility of the Information Mesh object system. A book paradigm will be studied to reveal fundamental concepts and insights for the overall development of a composite model. These fundamental concepts and insights suggest a general composite model as an extension of the Information Mesh. A book prototype has been built to demonstrate composite model and to expose implementation issues in details.

1.1 Motivation

A composite model allows the composition of a collection of Mesh objects into a single object. Many distributed information applications recognize composition as an essential capability. In a study of hypertext system, F. Halasz suggests that "the basic hypermedia model lacks a composition mechanism, i.e., a way of representing and dealing with groups of (objects) as unique entities separate from their components. [10]" His Dexter Model of Hypermedia System supports composition as a hierarchical structuring mechanism. [11] Furthermore, Halasz suggests that composition should be "implemented within, as opposed to on top of, all hypermedia systems. '[10]" Therefore, a hypertext model requires composite capability, and this capability should be implemented within hypermedia systems. Composition is necessary at the Information Mesh level.

Another motivation comes from the nature of the Information Mesh itself. [3] The Information Mesh is a distributed system, and may not be able to provide complete, system-wide, information on individual objects. With the lack of system-wide knowledge, each object may need to maintain its own information. Thus the need of a composite model partly comes from the distributed nature of the Information Mesh Model.

1.2 Goals and assumptions

Before designing a composite model, certain assumptions need to be made about the model. The following assumptions are based on previous studies on designing models for composition.

1. A component should not be aware of its containing objects. Otherwise, the list of contained objects can be very large. The cost of maintaining a large list is undesirable. This constraint is stated in the Linking in a Global Information Architecture paper [2].

2. A component can be included in more than one composite.

3. A component is allowed to contain other components. Thus a restriction is needed to prevent any cycle in composite/component relationship, that is, a composite should not be a component, directly or transitively, of itself [10].

1.3 The Information Mesh Environment

The development of a composite model is a step in achieving the overall vision and goals of the Information Mesh project. The composite model needs to be consistent with the overall Information Mesh object system. We will first look briefly at the overall vision and goals of the Information Mesh, then at the object model.

1.3.1 Overall Vision and Goals

The overall vision of the Information Mesh Project is to provide a long-lived global architecture for networked-based distributed information reference, manipulation and access [3]. The architecture should provide a minimal set of requirements so it does not restrict the evolution of the network. The hope is that the Information Mesh will become the primitive abstraction around which applications are built [4].

The specific goals to meet this vision of the Information Mesh are:

- *Global Scope.*

  The Information Mesh should provide a general agreement on object referral in a highly scalable manner.

- *Ubiquity.*

  The Information Mesh should support all network-based applications wherever the information they access is located.

- *Heterogeneity.*

  The Information Mesh must support a broad set of network protocols and applications. The set includes past implementation and likely future implementation.

- *Longevity.*

  Information and its identifiers should be able to survive for at least 100 years.

- *Evolvability.*

  Future applications and a changing network may place new requirements on the Information Mesh. Therefore the Information Mesh must be able to evolve and adapt to these new requirements.

- *Resiliency.*

  The Information Mesh should be resilient to failure in accessing information. These failures may arise due to a variety of reasons such as hardware failures or expired pointers.

- *Minimality.*

  The architecture should place the minimum necessary restrictions on applications. Thus it must provide as few restrictions as possible to achieve the above goals.

1.3.2 The Information Mesh Object System

The Information Mesh object system provides a typing model that achieves its goals of flexibility and evolution. This typing model allows a data object to play a variety of *roles*; an object's type is defined by the set of *roles* the object is playing. Each data object has an external appearance entirely defined by the roles they play. The object's appearance may change from time to time depending on the set of roles the object chooses to play at run-time.

Roles are arranged into an inheritance hierarchy. When an object plays a particular *role*, it also plays all that *role*'s super roles. The root of the inheritance hierarchy is the *object role*. Any first class object in the Information Mesh must play the *object role*. Although the role hierarchy is rooted in the object role, it also supports multiple inheritance.

Each *role* specifies an abstract interface to a certain type of data object. These abstract specifications are distinct from their actual implementations. The interface includes *parts*, *actions*, and *makers*. *Parts* describe the abstract structures of the data object that plays the role. *Actions*, similar to traditional "methods", are specifications of functions that operate on the object. *Makers* define the abstract functions that create instances of the object.

1.4 Specific Problem and Proposed Solution

The *part* characteristics of the *role* interface gives rise to complications in developing a model of composite objects. A *part* specifies an interface to an object structure. Each *part* has a name and specifies how substructures are exposed. A *part's* name is called *part-name* and an object's actual substructure is called *part-instance. Part-instances* may overlap. The overlapping of *part-instance* is a difficult issue that needs to be addressed in designing a model of composite objects. This flexible substructure in *role*-interface will be integrated with a fixed storage structure in the model of composite objects. Thus the integration of the *part* component is a major issue in developing a composite model.

By studying the book paradigm, this thesis attempts to resolve the underlying problem in the conflicts between *part* and composite model. The underlying problem is the implicit assumption that a role's *parts* and components are related. Our proposed solution is to separate role implementation from composition mechanism. The composition mechanism is implemented in a lower-level role, using *part-instances* which will not overlap.

1.5 Roadmap

The remainder of this document will describe the composite model and the demonstration of the model in a book paradigm. Chapter two describes the design of the composite model. It presents a model specific to the book paradigm, and then proposes a general model for the Information Mesh. Chapter three describes the detailed design of the book prototype. This chapter divides the system into different modules, and describes each module. Chapter four describes the implementation of the book paradigm. Chapter five concludes this report and suggests future work.

Chapter 2

Design Overview

In order to understand the Information Mesh composite model, it is necessary to study an extended example of the model. This chapter studies a book paradigm, and generalizes from this paradigm. The generalization results in a composite extension to the Information Mesh Object Model. In the book paradigm, a book is composed of components. The components are distributed over the network. The paradigm exposes various design and implementation issues to the overall composite extension in the Information Mesh Object System.

The content of this chapter is presented as follow. Section 2.1 discusses the overall model of the book paradigm. Section 2.2 discusses the Composite Model as part of the overall Information Mesh Object System. Section 2.3 summarizes the design.

2.1 Overall Model of the Book Paradigm

In the book paradigm, the components of a book are distributed over the network. The components are limited to be *file* objects. The user interface is presented in the *book role*, which has *chapter* and *page* as its *parts*. Studying this paradigm exposes some underlying problems in developing a composition mechanism. The problems arise due to possible conflicts with the existing role object model. We will attempt to resolve the conflicts in two design alternatives. Comparison of the design alternatives yields the specific model of composition in the book paradigm.

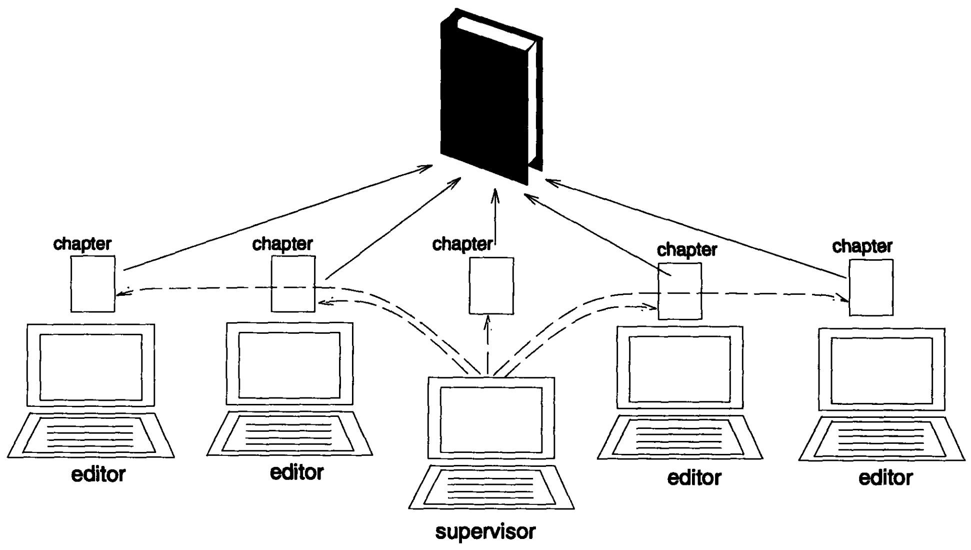


Figure 2-1: The example.

2.1.1 Example

Consider the following scenario depicted in Figure 2-1. A project team of five people is given the assignment of publishing an electronic book. The team has a project manager, who supervises the team. He organizes the book into five chapters, and assigns each team member(including himself) the job of writing one chapter. While each team member has editing power over only his own chapter, the project manager has editing power over all chapters. As well, they expect that a book can be divided into parts. It can be divided into pages, into chapters, or bibliography. Sometimes a team member may want to take out a whole chapter for reading or editing. Sometimes he may want to take out just one particular page to make a minor spelling correction.

This example raises a couple of concerns. First, different people may have different access rights over parts of or an entire document. In the Information Mesh Project, the security model is a orthogonal abstraction from the composite model.[1] Therefore the composite model should be free of security consideration. Though security is not the focus of this report, the composite model should be consistent with the security

[1]For demonstrating and efficiency purposes, we have not maintained this orthogonality in our prototype. This should not be taken as the preferred choice, but only expediency in the face of available technology.

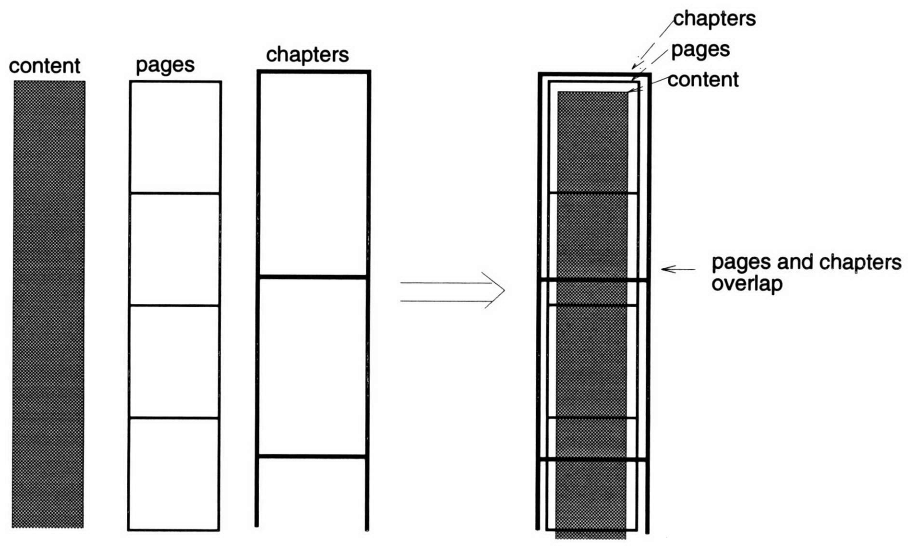


Figure 2-2: The flexibility of a role.

model in the Information Mesh. Thus in general, read and write access should be valid.

The second concern has to do with the flexibility of *parts* in a *role*. For example, *part-instances* may overlap(See Figure 2-2). A page may contain the end of one chapter and the beginning of the next chapter. As well, a *part-instance* may contain another *part-instance*. A page may be included in a single chapter. The composite model should not restrict the flexibility of *part* in the Role Object System.

2.1.2 Straight Forward Design of the Book Paradigm

In the most intuitive design, each component is either a chapter or a page. Consider a design which ignores the overlapping among *part-instances* as in Figure 2-3. Chapters represent the components of a book. Pages represent the components of a chapter. This straight-forward design serves as a basis. The basis will be modified to retain the flexibility in the role object system.

Consider the following example. The example reveals a problem in the initial straight-forward design. Suppose a person wants to read page 9 of a book. Page 9 contains the end of Chapter 1 and the beginning of Chapter 2. The top of the page

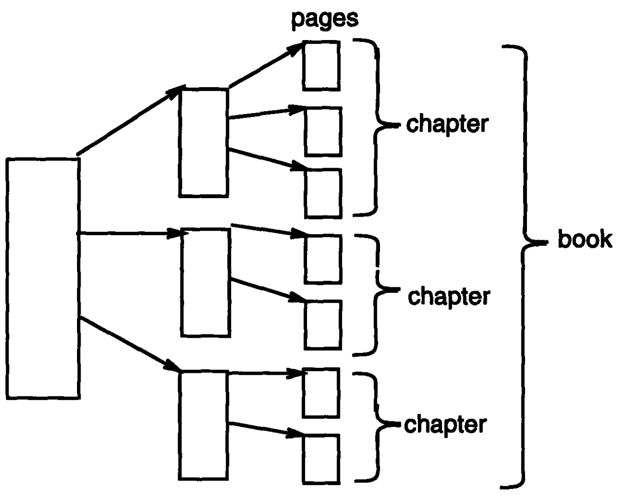


Figure 2-3: A straightforward design

exists as a component of Chapter 1. The bottom of the page exists as a component of Chapter 2. In this special case, if page 9 is kept as one component, then chapter 1 and chapter 2 point to the same component. They can no longer be distinct components(see Figure 2-4). To fix this problem, the straight-forward design is modified so that page 9 is composed of a top component and a bottom component. Chapter 1 and page 9 point at the top component. Page 9 and Chapter 2 point to the bottom component. Chapter 1 also points from page 1 to page 9. Chapter 2 also points from page 10 to the last page of the chapter. The book points at Chapter 1 and 2, and other chapters.

Modifying the design becomes complicated when we consider writing a *part-instance*. For example, if an extra line is inserted on page 10, the last line of page 10 is shifted to the first line of page 11 as in Figure 2-5. Lines are shifted in a similar fashion on every subsequent page. Changes would occur in all components with a higher page number than 10. Thus the insertion of a single line causes changes throughout the entire document. This replacement process is very inefficient. The inefficiency may slow down the overall performance. Moreover, the effect of an uncaught error is magnified by this inefficiency. An error in one component may propagate to many components.

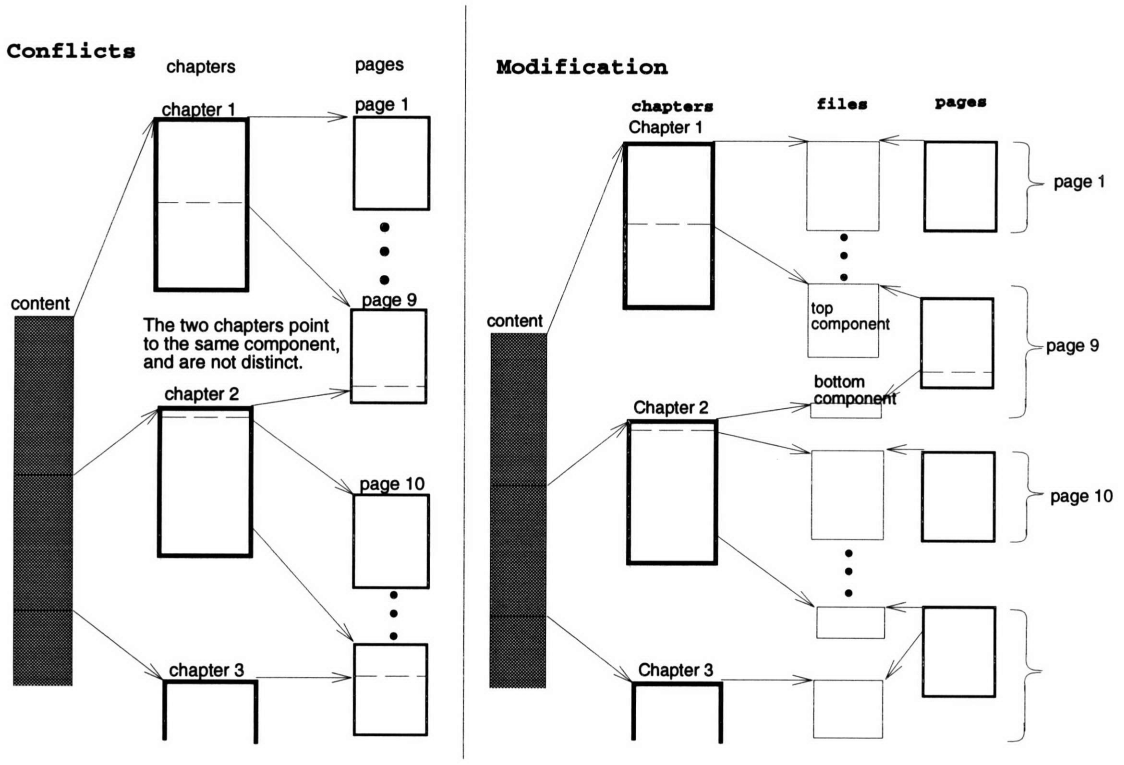


Figure 2-4: Conflict and Modification.

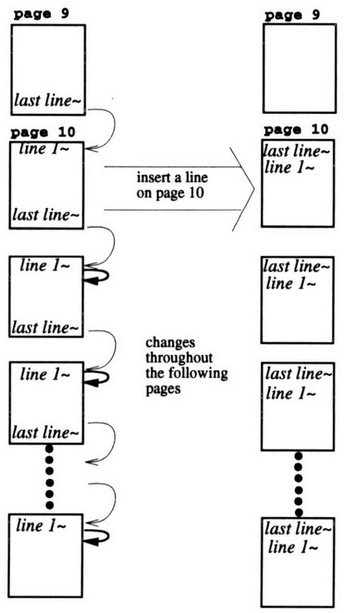


Figure 2-5: Problems in writing.

Most importantly, this replacement process violates a constraint stated in the Linking in a Global Information Architecture paper: "contained objects are not aware of their inclusion in a composite object. [2]" This constraint is violated in our example, in which changes in one component cause changes in other components. Without violating the constraint, the component has no way of informing its changes to its container, and hence to other components.

2.1.3 Hypothesis of no Association between component and part

The previous example can be generalized into the following hypothesis: *any association between parts and components may require components to be aware of their containers.* Another way of stating the hypothesis is: *any association between parts and components may require changes within one component to be reflected in another.* In this section, we will give an informal proof of the hypothesis. This hypothesis will support the second design of the composite model, which is presented in section 2.1.4.

In the informal proof, the association between *parts* and components is represented by an overlap of two sets(see Figure 2-6). Let *part-set* be a set that represent all elements of *part-instances* in a role. Let *component-set* be a set that represent all elements of components. The association between *part-instances* and components corresponds to the overlap of the two sets. For example, if chapters are components, then the *part-set* and the *component-set* completely overlap with each other.

To prove the above statement, we first realize that changes in one *part-instance* may cause changes to other *part-instances.* For example, by inserting a line on page 10, all subsequent pages need to be changed(see Figure 2-5). Thus changes in one *part-instance* propagate to changes other *part-instances.* Note that this propagation of changes cannot be eliminated, because it reflects the intrinsic nature of the page abstraction.

From the example in Figure 2-5, if a page is a component, then changes in one component cause the changes in others. The constraint is violated. However, a page

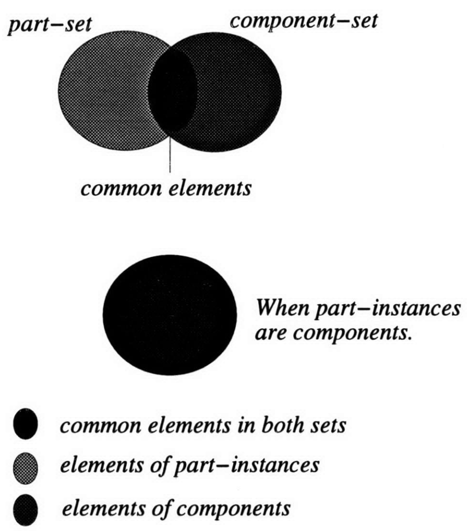

part–set
component–set
common elements
When part–instances are components.
common elements in both sets
elements of part–instances
elements of components

Figure 2-6: A part set and component set.

does not have to be a component to violate the constraint. The constraint may be violated if a component retains some properties of a page. Consider the elements of a *part-instance* which cause changes in other *part-instances.* It is possible that some of these elements are represented by the overlapping region between the *part-set* and the *component-set.* Then changes in one component may cause changes in other components. Therefore, the hypothesis that *any association between parts and components may require changes within one component to be reflected in another* is valid.

Moreover, even if the hypothesis is not valid, associations between *part-instances* and components still raise complexity. First, the implementation of one *part* may conflict with the implementation of another *part.* If a role has many *parts*, the relationships among *parts* and components become very complex. The complexity is even higher when an object plays multiple roles, because *parts* in different roles need to interact.

We have shown that *any association between parts and components may require components to be aware of their containers.* Moreover, any association between *part-instances* and *components* would increase the overall complexity and restrict extensibility. Therefore a composite model should not associate *parts* with components.

2.1.4 Design of the Composite Model

The second design removes all associations between *parts* and components. The composition mechanism is implemented in a composite module, which is responsible for combining objects. From now on, objects of the composite module will be called composite objects. Composite objects have the same interface as the objects which make up composite objects. They are wrapped around by book *roles*(See Figure 2-7). The book *role* is implemented in a book module, which is only responsible for exposing *part-instances*.

In the book paradigm, the composite module combines only file objects. Thus the interface of composite objects is similar to the interface of file objects. For example, the byte offset from the beginning of a file is called a **filepointer** in the Java programming language[2]. Thus, the byte offset from the beginning of a composite is also called a **filepointer**. While the composite module takes care of the composition mechanism, the book module implements the mechanism for accessing pages and chapters. Since a composite object is similar to a file object, implementation of the book module should be the same as the implementation before the incorporation of the Composite Model.

This design has the following advantages. First components do not need to know about the state of other components, because the design does not associate *parts* and components. Second, the abstraction between the book module and the composite module is very clean. The two modules have distinct functionality. The book module finds pages and chapters. The composite module combines objects. On the other hand the performance may not as good as the straight-forward design, especially if *part-instances* rarely overlap.

We choose this design to build the book paradigm. Building the book paradigm will be discussed further in Chapter 3 and 4. The remainder of this chapter attempts to generalize from this design and extends the Information Mesh to support the general composite model.

[2]We choose Java as our programming language. The choice is discussed in Section 3.1.

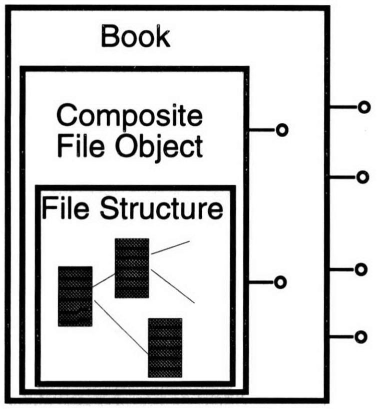


Figure 2-7: A book wraps around a composite.

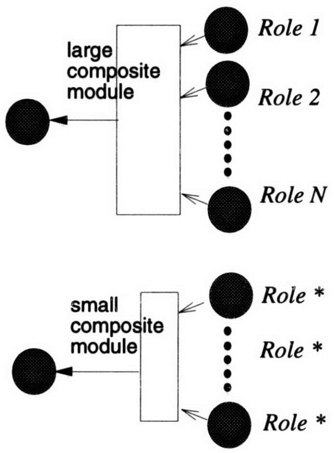


Figure 2-8: The first issue.

2.2 Extending the Information Mesh Object System

The previous section describes a specific design for the book paradigm. The design wraps around a collection of files with a composite abstraction, and builds a book interface on top of the composite abstraction. In this section, the design will be generalized so that any objects can be combined in the Information Mesh.

In coming up with the general composite model, we need to consider two issues. The first issue is the number of *roles* which a composite module can combine. In the book paradigm, the composite module only combines objects of the same role. In

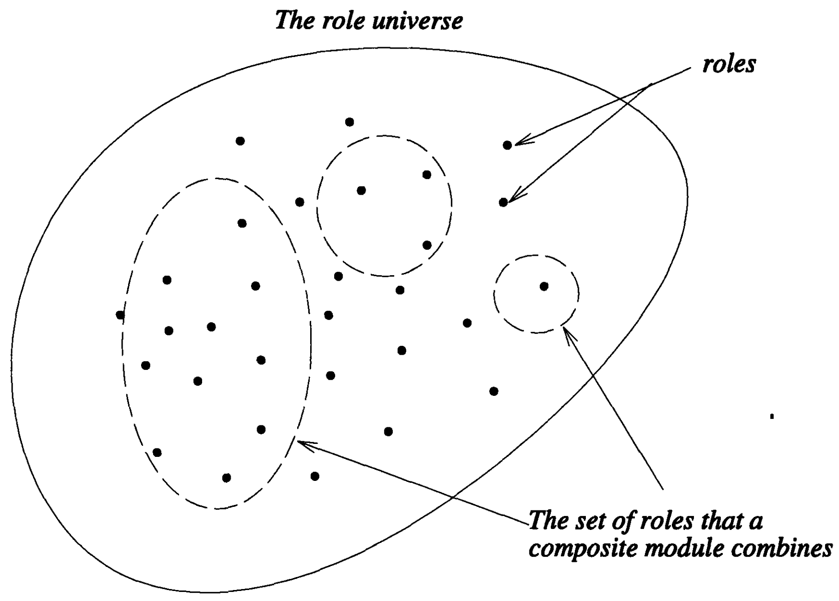


Figure 2-9: Sets of roles.

general, a composite module may combine objects of different roles(see Figure 2-8). The largest composite module combines objects of any roles. The smallest composite module combines only objects of the same role. The second issue is the representation, or the nature of a composite module. For example, a composite module can be treated as part of an implementation, or it may be represented in a *role*.

2.2.1 First Issue

A composite module may be designed to combine objects playing different roles. Let us denote the *size* of a composite module to be the number of roles it can combine. The first issue involves the appropriate *sizes* of composite modules. The Information Mesh may have a number of composite modules. Each composite module combines a certain set of roles(see Figure 2-9). For example, in the book paradigm, the composite module combines objects which play only the file role. The following compares a design of large composite modules with a design of small composite modules.

In a design of large composite modules, a large variety of roles can be combined. For example, consider a composite module of *size* 8. The module can combine ob-

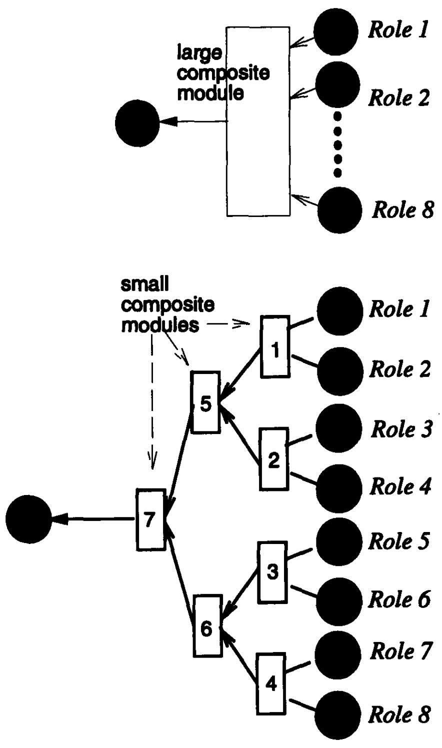


Figure 2-10: Example.

jects of 8 distinct roles. In contrast, consider composite modules of *size* 2. They cannot combine objects of 8 distinct roles. In order to combine objects of 8 distinct roles, 7 composite modules of *size* 2 are needed(see Figure 2-10). A design of small composite modules has another disadvantage. With small composite modules, it is often necessary to add new modules to combine a specific pair of objects. But an environment with many composite modules may be confusing to programmers. When programmers want to combine objects, they may encounter difficulties in finding the suitable composite modules.

We should also consider the complexity issue in a composite module. There are different factors which may increase the complexity in a composite module.

On the other hand, a large composite module may be more complex to implement than a small composite module. There are a number factors which may increase the complexity in a composite module. For example, the similarity and dissimilarity of the roles involved will have a strong impact on complexity. If the roles being composed are very similar, the composite module will be simpler. The *size* of a composite module should also have a strong impact on complexity. For example, consider a composite module of *size* 3. The module must handle the differences among the three combining roles. The differences may exist in the *actions*, *parts*, or *makers* in a role. Suppose the module is modified so that it will be able to combine a fourth distinct role. The new module needs to handle three more relationships. It must handle the differences between the new role and each of the three original roles. Therefore the modified module will be more complex than the original module. In general, a large composite module is more likely to have higher complexity than a small composite module. A large composite module may also be slower than a small composite module because of the extra complexity. Suppose we want to combine objects of two roles, and two composite modules can do the job. Then the smaller composite modules may be more efficient than the larger module.

Looking at the relative advantages of large and small composite modules, we come up with the following design.

The General Design of the Composite Model

Roles are defined hierarchically in the Information Mesh. In the general composite model, the condition for composition is the existence of a common super role among the objects. Objects can combine only if they have a common ancestor role which allow its objects to be combined. To combine a collection of objects, we search the role hierarchy. After we find the closest common ancestor, we check if objects of this role can combine. If they can combine, we return an object which plays the common ancestor role. This returned object is a composite object in reality. Notice that the farther back we search in the hierarchy, the fewer similarities exist between the composite object and the original components. Then it may be difficult to transform the composite object into a desirable form. Therefore a closer common ancestor is preferable.

As an example of this general model, consider an **I/O stream role, a tape role**, and **a file role**. The **I/O stream role** is a parent of **a tape role** and **a**

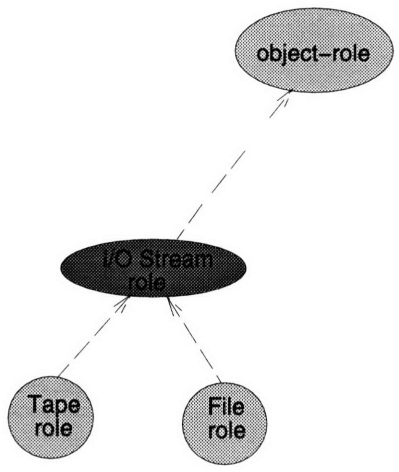


Figure 2-11: Example.

file role(see Figure 2-11). To combine a tape object and a file object, each will need to be considered as an I/O stream object. After the objects are combined, the composite object will play the I/O stream role. The I/O stream role has actions and parts common to the file role and the tape role. The I/O stream composite may be transformed back to a file object.

This design eliminates the disadvantage of the implementation complexity, while retains the advantage of a large composite module. The implementation complexity is completely eliminated, since each composite module is implemented as if it combines only one role. At the same time, the actual *size* of a composite module is equal the number of descendants of the combined role. Therefore, a composite module is very large if it combines a role with many descendants. For example, if the I/O stream role has a lot of descendants, and a composite module for I/O stream role can combine many other roles. This design also has an appropriate abstraction for composition. To combine two objects, the two objects must be similar in their actions and parts. But similarities among objects are also captured in the inheritance relationship. Therefore, by finding the closest common ancestor, we have identified the largest set of common actions and parts.

On the other hand, this design may be unable to combine some objects of similar structure. For example, a string object and an array object have very similar struc-

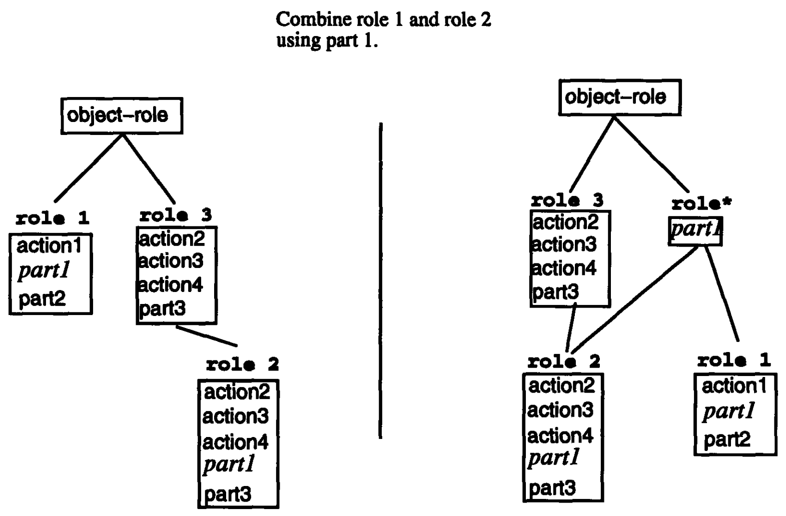


Figure 2-12: Example.

ture. Yet in a programming language such as Java, they have no common ancestor except the root object class. Therefore, if we want to combine them in the Information Mesh, they must be designed to have a common ancestor in the beginning. Under this design, we may also encounter problems if we initially build a few straightforward roles, and later want to combine some of these roles. For example, look at Figure 2-12. Suppose **role 1**, **role 2**, and **role 3** are initially designed as shown on the left-hand side. Later, we want to combine **role 1** and **role 2**, and present the composite in a role with no action and the single part **part 1**. Under this design, it may be necessary to rebuild **role 1** and **role 2** so that they derive from the common ancestor **role***. The redesign is shown on the right-hand side of Figure 2-12.

A possible design alternative is to identify common relationships among roles by the common actions and parts instead of by a common ancestor. However, actions and parts in two roles may have the same specification but very different behaviour. Therefore it is difficult to define which are the common actions and parts and hence a common ancestor is the only clear indication of common relationship among roles.

The general composite model allows objects to be combined after first converting the objects into a closest common super role. The closer the common ancestor, the more similar are the roles of the composite and components. A composite module can combine objects which may play any of the descendant roles. The implementation is still simple because all composite modules are built to combine objects playing one role.

2.2.2 Second Issue

Section 2.2.1 discusses the general composite model. The central issue is the *size* of each composite module. Section 2.2.2 will focus on the representation of these composite modules in the Information Mesh. There are different options for realizing composite modules. For example, composite modules may exist as part of an underlying representation in some roles, or they may also exist as composite roles in the Information Mesh Object System.

Our design chooses to represent composite modules as an underlying representation of some role. In the object role hierarchy, there can be a number of roles which support composition. For each of these roles, there are multiple implementations that allow for different underlying representations. One is the simplistic, straight-forward, non-composite representation. Another is the composite representation. The composite representation handles the composition mechanism. Its implementation is defined recursively, as it handles components playing the same role.

For example, consider the **IO Stream** role(see Figure 2-13). The **IO Stream** role is supported by two underlying representations. One representation is the straight-forward **IO Stream** implementation. The other is a composite representation, whose components play the **IO Stream** role. With this recursive definition, the recursion ends at a component which is a straight-forward **IO Stream** object. However, we must be careful that some composite representations may loop forever, and we must restrict the existence of such representations. For example, suppose *object 1* contains *object 2*, which in turns contains *object 1*(Figure 2-14). The mutual containment continues forever, and there is no end to the recursion(Figure 2-15). Then the composite is infinitely large. This infinite loop exists whenever a closed loop exists between two nodes of combined objects. Therefore we must make sure that the instance containment relationship forms a DAG(directed acylic graph).

A role which supports composition is considered identical to its non-composite

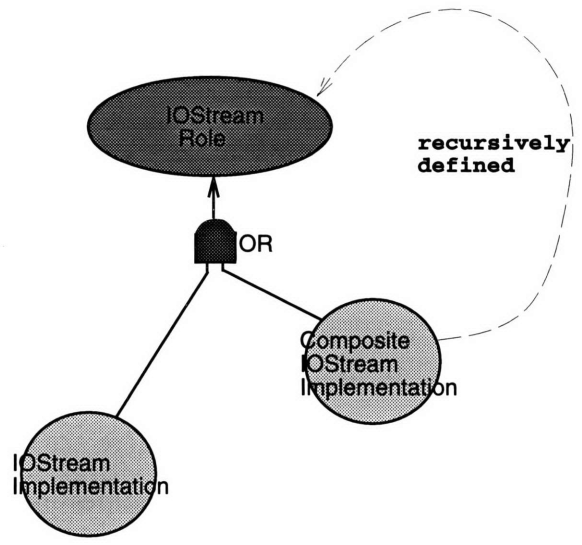


Figure 2-13: Example.

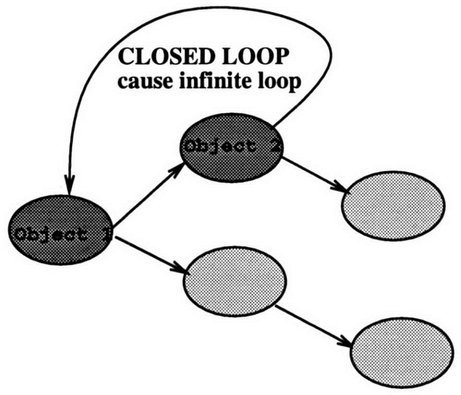


Figure 2-14: Closed loop.

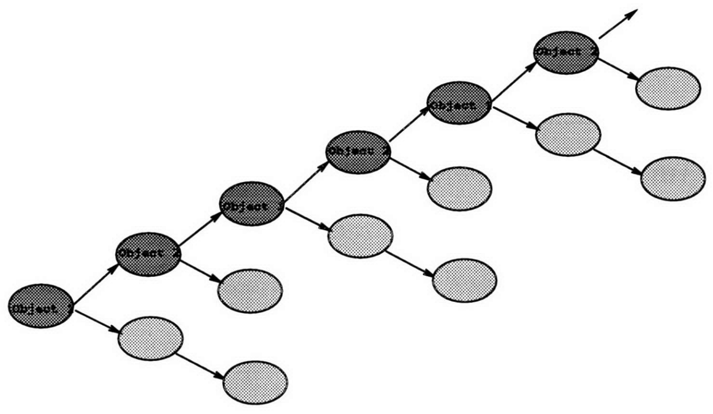


Figure 2-15: Recursion is expanded infinitely.

version. The composite version supports a few more optional actions to specify the parameters for inserting objects. For example, a composite **IO Stream** role should have all the actions and roles of a straight-forward **IO Stream** role. It should also support a set of optional actions to specify parameters for inserting an **IO Stream**. A parameter may be the *oid* of an **IO Stream**, or the exact location for inserting one **IO Stream** into another **IO Stream**.

In this design, the Information Mesh is built so that the composition mechanism in one set of roles is not exposed to the implementations of other roles. The abstraction is very useful since the development of the Information Mesh is divided into different areas. Therefore developers in one area only need to be concerned with the composition mechanism inside their area. For example, developers who build the naming service may use kernel roles, some of which may support composite. These developers do not need to understand the composition mechanisms inside these kernel roles.

In addition, a role can be modified easily to support composition. In some situations, the need for composition may not be recognized until a considerable amount of time has elapsed. Then, the original versions of certain roles need to be modified in order to support composition. In this design, the modification does not affect other implementations. The new role is modified to include an underlying composite representation and a set of optional actions to specify the composite parameters. These changes do not affect other implementations which have been using the role, because they see an identical interface in these roles.

2.3 Summary

This chapter has described the composite model in the book paradigm and the general composite model as an extension of the Information Mesh.

By studying the book paradigm, we discovered the conflicts between *roles* and composition. We developed a composite model which resolve this conflict. The model dissociates the composition mechanism from the module which provides the book interface. The book interface wraps around the composite module, which combines files into a single virtual files.

This model is extended to a general model in the Information Mesh. In the extension, certain roles support composition. To combine two objects, we find their closest common role ancestor which supports composition. These roles have multiple underlying representations. One is non-composite. The other is composite. This design allows composition to be possible for many roles. At the same time, it prevents the object system from getting complex.

Chapter 3

The Book Paradigm System

This chapter focuses on the design details of the book paradigm system we build and study. This book paradigm system is a software prototype which simulates the composite model. Section 3.1 explains our choice of platform and programming language. Section 3.2 gives an overall view of the client-server interaction. Section 3.3 describes the client-side design. Section 3.4 describes the server-side design.

3.1 Platforms and programming Language

The Java programming language is chosen to build the book paradigm, and Windows NT is the underlying operating system. There are several reasons this choice. First, Java is object-oriented. Therefore the whole system can be built using objects. This is important since the Information Mesh universe is composed of objects. Second, Java supports interactions with Web browsers. So page and chapter instances of a book can be displayed on a web browser. Third, Java is "architecturally neutral", allowing code to be portable and "is secure to survive in the network-based environment [8]." On the other hand, Java may have worse performance than C++ because Java is an interpreted language. However, the performance of Java should be adequate for most applications [8]. Since the overall performance is not the primary issue in building the prototype, the performance of Java should be sufficient for this project.

The book paradigm can be presented in different ways. It can be presented in a Web browser, or it may exist as a completely new standalone application. As a commercial application, building a new standalone application may be more attractive. However, building a new application may introduce many issues which are outside the scope of the Composite Model. For example, a book can be presented in HTML(Hypertext Markup Language) on a web browser. In a completely new application, a new way of document presentation needs to be designed. Therefore the book paradigm will be presented in a Web browser.

For the book paradigm to be presented in a Web browser, we can either modify an existing web browser or build a Java applet. Modifying an existing web browser is an impractical option because there is too much installed base. As well, there are currently many different web browsers in many different versions. Therefore a Java applet is the choice to present the book paradigm. On the other hand, security restrictions on applets do not allow reading or writing on local disks. These security restrictions may give rise to some fundamental differences between an applet model and a more realistic model of a new standalone application.

3.2 Client-Server Interactions

The previous section established Java as the programming language for the book paradigm, and the Java applet as the choice of presentation. This section examines the possible mechanisms which allow an applet to access a user specified *part-instance*. The choices of mechanisms are limited by the set of interactions which a Java applet provides. The Java applet restricts reading and writing on local disk, and can only initiate network connection with the machine it comes from. Based on these security restrictions on an applet, there are three possible approaches for accessing an object on a remote machine. First, an existing protocol, such as http or ftp, can be used to retrieve documents. Second, a private protocol can be designed, allowing an applet to communicate with the machine it comes from. An applet and a server process may open a socket using the Java Socket package. Then they may exchange string messages. Third, the access object is treated as a remote object, and remote functions

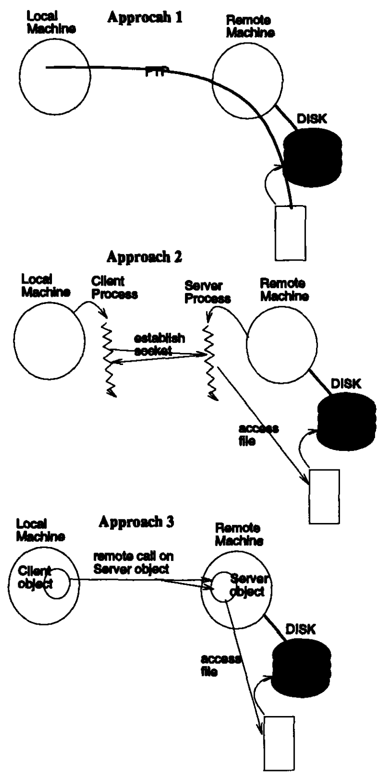


Figure 3-1: The interactions between a client and a server.

on the object are called using an existing RPC product. Figure 3-1 depicts these three approaches.

In the first approach, an existing protocol, such as ftp, is used to retrieve files. The file can be thought of as an object instance. The "object instance" resides at the local machine after retrieval. The "object instance" is a copy of the original object at the remote machine. Then the applet can modify this "object instance" at the local machine. To make permanent changes to the file, the modified "object instance" needs to update the original file on the remote machine. This updating mechanism may be implemented using the second or third approach. This first approach has the disadvantage that existing protocols only transfer files. Even though transferring files may work well in the book paradigm, this approach may not be generalizable to the composite model as an extension of the Information Mesh Object System. Therefore the first approach is taken out of consideration.

Both the second and third approaches involve interactions with a server process.

The server process is running at the remote machine from which the specified document originates. In the second approach, a private protocol is developed between the applet and the server process. The applet and server process are connected by opening a socket using the Java Socket class. Then, the applet and the server process communicate by exchanging string messages. The private protocol is in the same level as HTTP, NNTP, or FTP, but is much simpler. It is designed only to suit the specific needs in the book paradigm. This approach has the following disadvantages. First, it takes time to develop a robust protocol. Errors in protocol may be sensitive to the timing of each message. Second, if new functionality is added to the server, the protocol needs to be changed. The protocol needs to be well designed so that it can be easily modified and extended.

The third approach is much simpler to implement. Approach 3 involves technology in distributed computing such as CORBA. An applet can invoke methods on a remote object at the server machine. First, the applet accesses a remote object through some naming service, and remotely invokes a method on this remote object. The interaction with a server process allows a client applet to read or write on a remote object, because the server process has the authority to read and write any files in the server machine's disk. This approach is simpler to implement, and is a more elegant design. Moreover, adding new functionality is easier than the second approach. Only the common interface, between the applet and the server process, needs to be changed.

Therefore the third approach is clearly superior to approach 2. There are two other concerns, however. The first concern is the performance issue. The second concern is the choice of current technology in distributed computing. The first concern is that the performance of the second approach may be better than the performance in third approach. However, the difference in performance is likely very small. The second concern is the choice of available technology which allows distributed computing. Currently, there are many commercial CORBA ORBs available, but they are all very expensive, and provide significantly more services than this specific project requires. However, there are freeware options which provide the technology of distributed computing. Moreover, they are all specialized for the Java programming language. Among the freeware products that support remote object invocation, there are Java IDL [14], Java RMI [15], and HORB [16]. Java RMI is probably the most efficient and easiest to use. However, the netscape browser does not support it yet. Netscape does support Java IDL and HORB. HORB supports only with Java JDK 1.0.1, but not Java JDK 1.0.2. Therefore Java IDL was chosen for this project.

The basic interaction between client and server is as follows. At initialization, the client applet finds a server process that can create a book instance. When the applet asks the server to create an actual book-instance, the server process creates the book instance. The book instance exists as an object instance at the server site, not at the client site. The server process binds the book instance to a unique name using functions supported by Java IDL. The server returns the unique name to the client applet. Using this unique name, the applet asks the Java IDL's ORB to resolve the book-instance. The applet then creates a stub for the book-instance. Then, the applet can call different functions on the book stub based on the interface of the book role.

3.3 Client-Side

The client side handles the user interface. The goal of the user interface is to demonstrate that underlying mechanisms function properly. The user interface may not be the most user-friendly because the focus of this project is on the Composite Model. With this goal in mind, this section presents the user-interface, and its interaction with server objects.

3.3.1 User Interface

As depicted in Figure 3-2, the user interface begins with a document with two frames on a web browser. The top frame contains an applet. The bottom frame displays a page or chapter. The applet allows a user to request a specific page or chapter. The bottom frame displays the requested page or chapter.

The applet has two text field entries. The first entry allows a user to specify

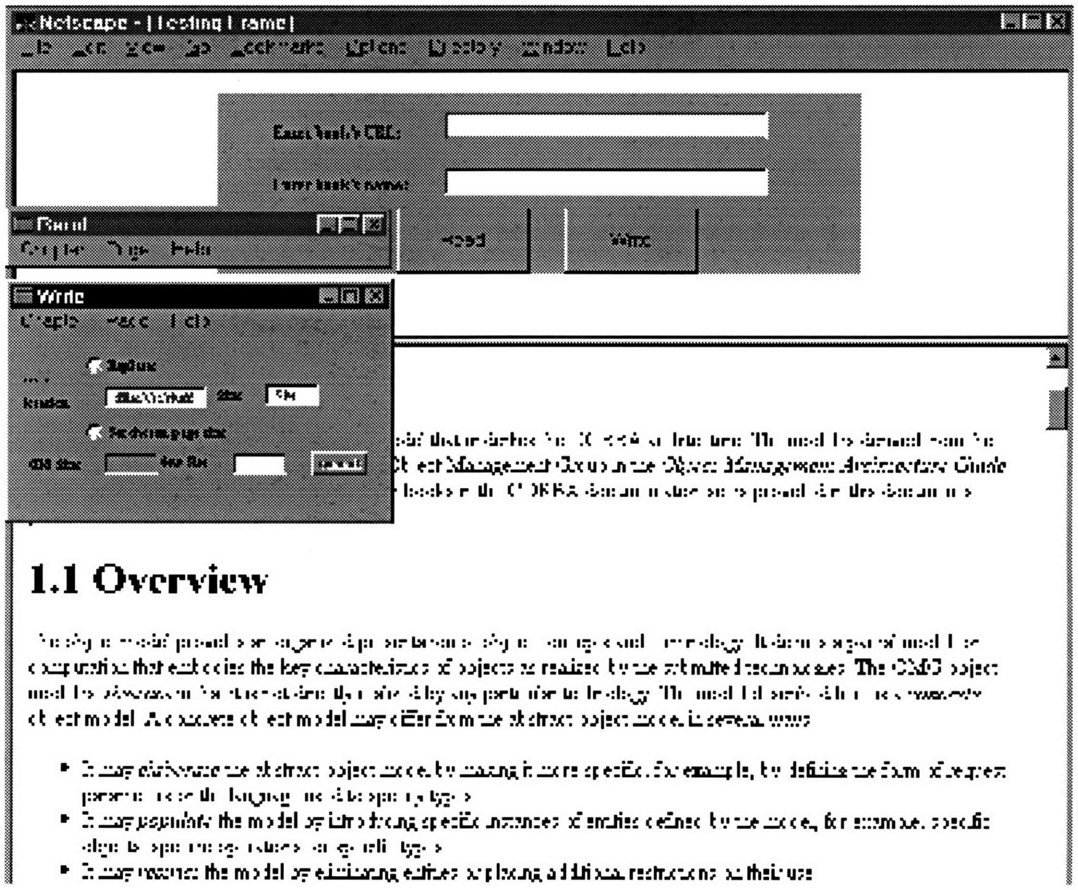


Figure 3-2: User interface.

the URL of a book. The second entry allows a user to specify the name of a book. The applet also contains a "READ" button and a "WRITE" button. Invoking either button causes a new window to appear. If the "READ" button is invoked, a new "READ" window lets a user request a desired page or chapter, which appears in the bottom frame. If the "WRITE" button is invoked, a new "WRITE" window allows a user to specify how to edit the book.

Each new window contains a menu. The menu in the "READ" window lets a user choose specific pages and chapters to be displayed. A user chooses specific pages and chapters by clicking their corresponding indices in the "READ" window. The menu in the "WRITE" window lets a user modify specific pages and chapters. A user can modify pages and chapters in two ways. The first way is to replace them with new files. In this "Replaced" option, the "WRITE" window has a textfield entry for indicating the URLs of new files. Pages and chapters are replaced when a user clicks the corresponding indices on the menu. In the second way, a user modifies pages by setting their sizes. In this option, the "WRITE" window has two textfields and a "commit" button. The first textfield displays the original size of a page. The size, in bytes, appears in the textfield after a user clicks on the desired page index on the menu. The second textfield allows a user to specify a new size for the page. The "commit" button lets a user commit the changes permanently.

The user interface can be designed to be more user-friendly. For example, instead of creating new "READ" and "WRITE" windows, the applet may contain the read and write menus. However, Java does not allow menus to be inside an applet; menus can only appear in a new window. In addition, in a more realistic application, there is no need for the URL and name entries on the applet. Instead, a user would directly access the book by a call to a Netscape browser. In this case, the page would contain an invisible applet for creating a menu window. However, our applet is designed for testing and debugging purposes. To access another book, one can just change the entries in the applet. Thus the current design is chosen over designs which are more user-friendly.

3.3.2 The Client Object Model

When the applet page is browsed, invocation of the "READ" or "WRITE" button creates a menu window. In our design, each initialization of a menu window also corresponds to a book-instance creation. Thus each menu window represents a separate copy of a book-instance(See Figure 3-3). The abstraction is similar to the books in the real world where there are many copies of one publication.

There may be a better object model at the client site. For example, it might be better if only one book instance existed at any time(See Figure 3-3). However, there are many issues involved in the object model. For example, should readers of a book be notified of updates to the book in real time? If so, a reader may skip a section while someone else is editing the book. For example, suppose a reader is reading page 9. At the same time, someone else deletes a line on page 8. Then the reader turns to page 10. The first line on page 10 becomes the last line on page 9, and the

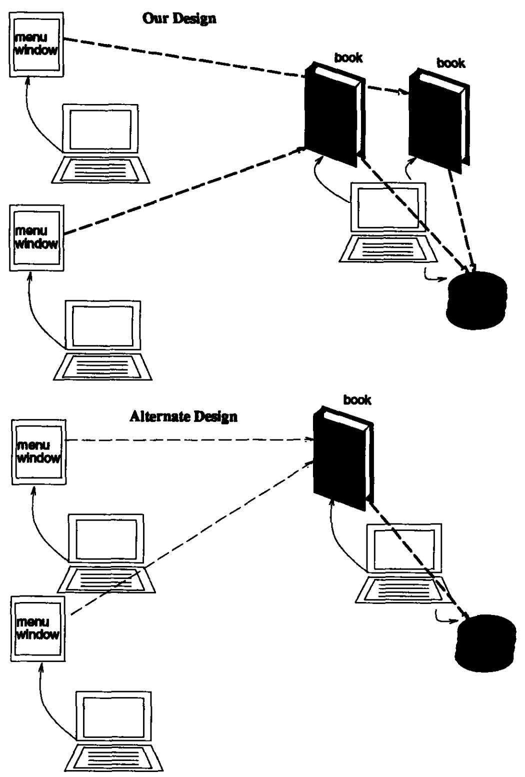


Figure 3-3: Object representation.

reader would miss reading that line(See Figure 3-4). There is a locking mechanism needed here, although we will leave locking as a topic for later study. We will deem the current design of the client object model sufficient for now.

3.4 Server-Side

This section discusses the server side design. On the server side, the Book class and the Composite class are distinct modules. At run-time, instantiation of the Composite class will be wrapped by the book abstraction.

The section starts by discussing the detailed design of the Composite class, followed by the detailed design for the book class. Finally, the issues of persistent object storage is studied.

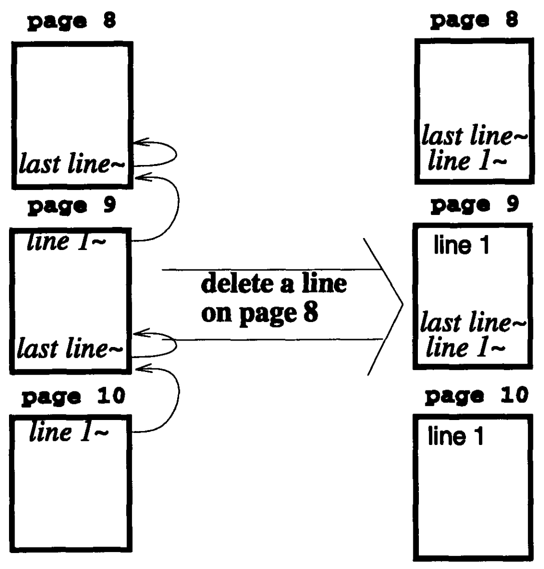


Figure 3-4: Example.

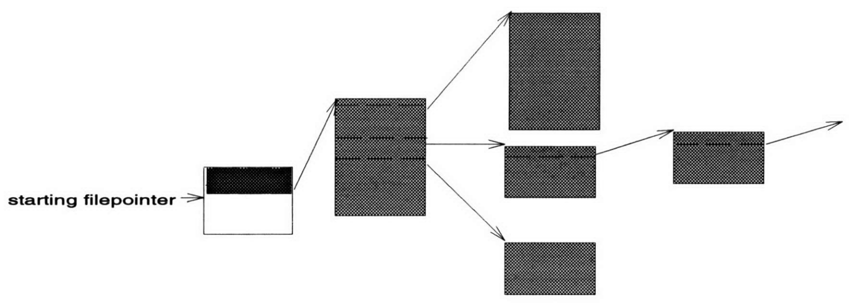
3.4.1 Detailed Design of the Composite Class

The Composite class has an interface that makes it appear to be a single file. The class has a read method which reads the content of composite objects. The read region is specified by a starting and an ending filepointer. The class also has a write method which writes onto Composite objects. The replaced region is also specified by a starting and an ending filepointer.

In the simplest design of the Composite class, insertion of a component is indicated by special tag. Then to find the starting filepointer, the composite object needs to be scanned from the top. When a special tag is parsed, the scanning switches to the component. If the component is finished, the scanning switches back to the container. A counter is incremented each time a byte is scanned. When the counter is equal to the starting filepointer, the composite object can start to read or write.

This design is similar to how LATEX and HTML work. They keep the amount of information on the files' relationships to a minimum. This advantage allows implementation to be more straight forward, since there is no need to manipulate any information on files' relationship.

On the other hand, this design is very inefficient. Consider the case in which a



▨ – the region which does not have to be scanned

■ – the region which needs to be scanned

Figure 3-5: Inefficiency in reading a complex file structure.

document contains very large components. If the starting filepointer is below the insertion of a component, then the whole component is scanned through unnecessarily(See Figure 3-5). The inefficiency is even greater when there are many levels of components. However, this inefficiency can be prevented if the information on component sizes is available. By keeping track of component sizes, high-level virtual filepointers in a composite can be translated into real filepointers in disk files. Then, the starting filepointer can be found without traversing levels of components.

But then, why do LATEX and HTML use this design to include components? To answer this question, we first realize that LATEX and HTML are high level markup languages. They convert the content from text files into the correct format and document presentation(See Figure 3-6). Even without composition, their mechanisms need to read through the entire document because they are context sensitive. But the Composite class is at a lower level abstraction. Its only function is to combine files into a virtual file. The class is not concerned with how the virtual file will be presented. Therefore this design causes inefficiency in the composite class, but not in LATEX and HTML.

In our design, each insertion of a component is associated with the following information:

- the URL of the component.
- the filepointer for inserting the component.

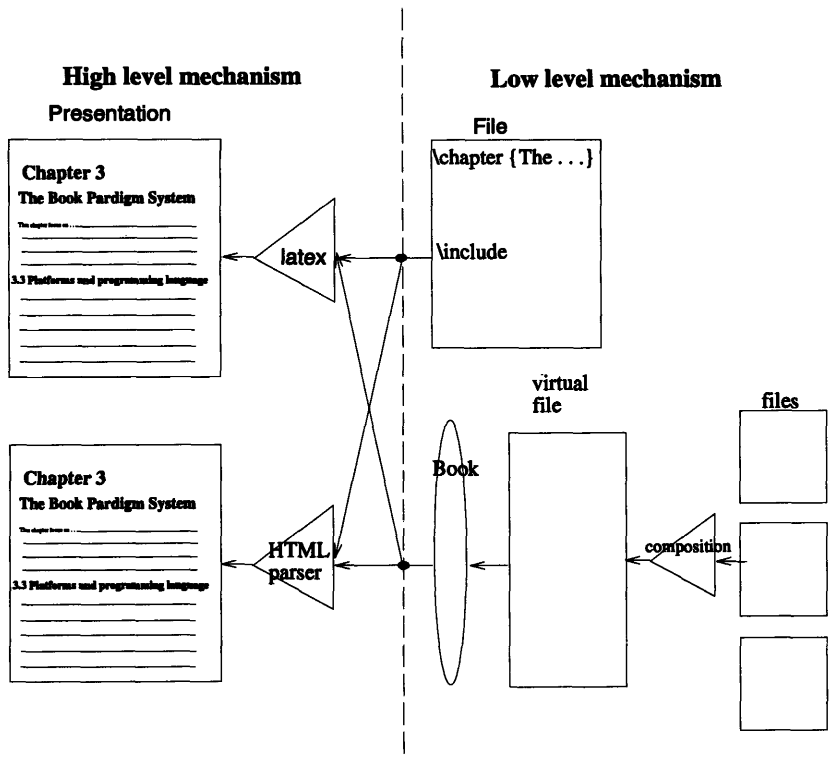


Figure 3-6: Different levels of mechanisms.

- the size of the component.

The disadvantage of this design is the potential complexity in maintaining and updating the above information. The complexity is further discussed in Chapter 4, which describes the implementation of the book paradigm. As well, in this design, the insertion filepointer does not shift with the content, even though the insertion of a component should shift along naturally with the content(See Figure 3-7). For example, suppose the sentence "I play tennis" is composed of "I play ", and "tennis". The offset for "tennis" is 7 in the sentence. Then "I play " is changed to "I played ". Thus the new sentence should be "I played tennis." The offset of "tennis" needs to be updated from 7 to 9. Therefore this design requires an extra mechanism to shift all insertion points. The extra mechanism creates more complexity. Thus the design may not be optimal.

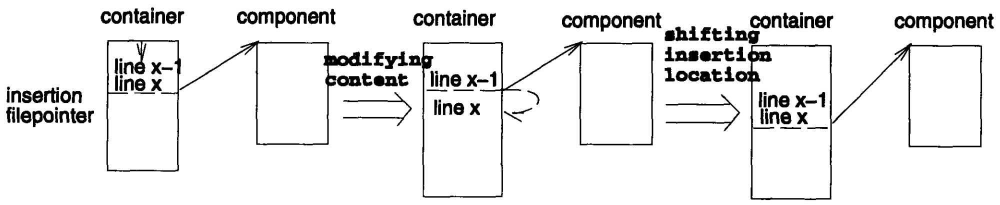


Figure 3-7: The shifting of insertion filepointers.

3.4.2 Detailed Design of the Book

This section first specifies the book role, which is the interface of the book. Then it describes the object model at the server. Finally, this section describes the chaptering and pagination in the book role implementation.

Book Role

The book role provides functions that define a book interface to users. The core actions in the book role are: retrieving a page, retrieving a chapter, replacing a page, and replacing a chapter. Other actions such as maximum page, maximum chapter, page size, and setting page size, are also useful to the client applet. The only parts of interest for this study are page and chapter.

The Object Model in the server machine

For the object model in the server machine, each book instance is created by a remote call on a Server object. The Server object associates the book-instance with a composite object. All these object instances reside at the server machine(See Figure 3-8).

In an alternate design, book instances and composite objects reside at the client machine(See Figure 3-8). This design is possible since a Java applet can load remote classes. Then, remote invocations are necessary at a lower level in order to modify disk files at a remote server machine. Thus, one possibility is to design a Server object which has the ability to write files and allows its methods to be invoked by remote objects.

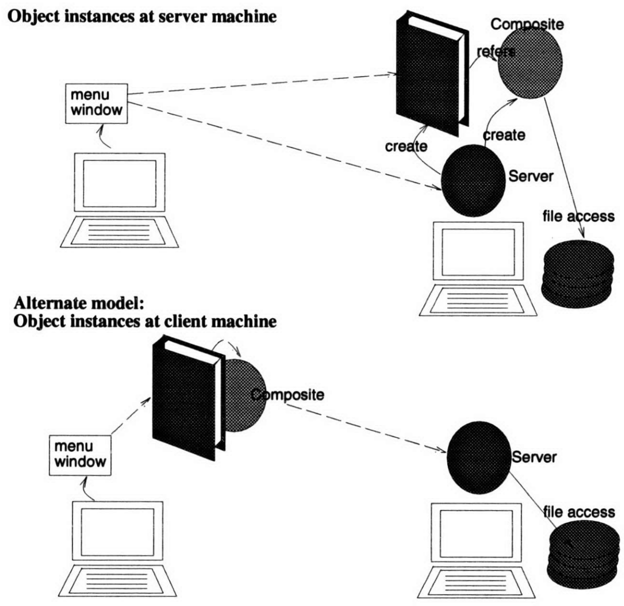


Figure 3-8: Object Model at the Server machine.

Pagination and Chaptering

The main function in the book class is in retrieving a requested page or chapter. The book class retrieves a part-instance by locating its beginning and its end in the composite. There are two possible mechanisms for locating the beginning and the end. The first mechanism scans through a document and parses for a page break or chapter break. The second mechanism uses a list of filepointers. Each filepointer indicates the beginning of a part-instance, and the end of the previous part-instance. The mechanism for pagination is first discussed, followed by the mechanism for chaptering.

In our design, pages are retrieved by the second mechanism, where a persistent list of filepointers refers to the beginnings of pages. The second mechanism is superior to the first mechanism in the following way. Consider the example in which a line is deleted on *page x*. The first mechanism needs to shift the page breaks for all pages following *page x*. Shifting page breaks is complicated because it involves deleting and inserting page breaks in the content. On the other hand, the second mechanism does not need to modify the list of filepointers. This list of filepointers acts as fixed boundaries on top of the content(See Figure 3-9). The fixed boundaries are unaffected when the content is modified. Therefore, as a line is deleted, the shifting in pages

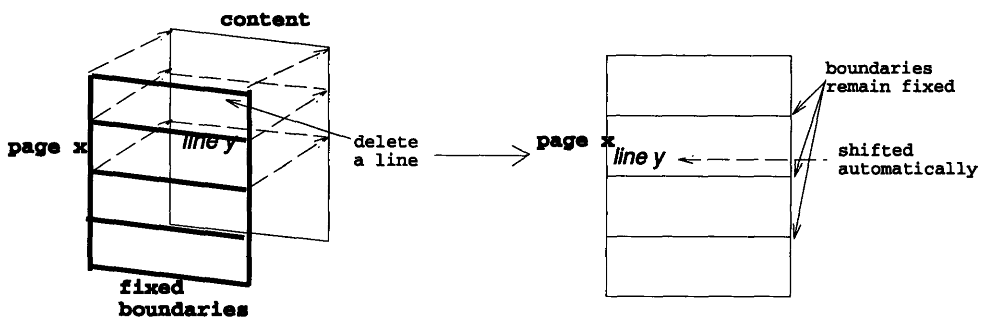


Figure 3-9: List of page filepointers as fixed boundaries.

propagates automatically to the following pages. The second mechanism reflects the nature in the page abstraction. The page abstraction separates a document into sections by sizes, not by content.

On the other hand, there is the concern that page sizes may not be identical. For example, some pages may contain larger headers and consist of fewer bytes than other pages. For example, the page which contains "Chapter *X*" likely consists of fewer bytes than the following page. With this concern, the second mechanism is still superior to the first mechanism for two reasons. First, shifting page pointers is easier than the first mechanism. It involves incrementing and decrementing numbers. The second reason is that the book class is implemented in a separate application which is also responsible for the actual presentation. For example, the application may be similar to Latex. In this separate application, the size of a page may be determined by some other units than bytes. The units reflect the size in the actual representation. For example, an empty line has the same value as a line of text, because both lines take up the same space in the actual presentation. As well, a character of 18pt font is larger than a character of 16pt font by their difference in size. Using this unit, all page sizes are identical.

In our design, chapters are also retrieved by the second mechanism. The initial design chooses the first mechanism, but the second mechanism is found to be more practical. In the initial design, chapters are indicated by chapter breaks embedded in the content. Since chapter is an abstraction which separates a document by its content, the initial design may be better because one chapter is unaffected by changes in another chapter. On the other hand, the second mechanism needs to manipulate chapter pointers to reflect changes in another chapter. For example, suppose chapter 1 is modified and becomes larger. The starting filepointer of chapter 2 needs to be updated to be a larger filepointer.

However, the initial design has the difficulty of finding a specified chapter tag. In one possibility, the mechanism scans through the composite object and parses for the specified chapter tag. Scanning through the entire composite object is inefficient and may slow down performance. Another possibility is to modify the design of the composite class. Then a region in a composite object can be indicated by a beginning chapter tag and an end chapter tag. However, this idea breaks down the abstraction model, since the composite class is different from a simple file with the knowledge of chapters. On the other hand, even though the second mechanism needs to manipulate the list of filepointers, shifting filepointers is easy. It involves incrementing and decrementing numbers. Therefore chapters are retrieved by the second mechanism.

The current model exclusively relies on filepointers. Both chaptering and pagination use a list of filepointers to indicate the starting position of each part-instance. Manipulation of the list is often necessary to keep the list consistent with changes in the book.

3.4.3 Persistent Storage

Section 3.4.1 and 3.4.2 describe the designs of the Composite class and the Book class respectively. Both designs contain structures that contain persistent data, such as the URLs of components. The persistent data can be stored in several representations. They can be stored in text files, a Java class called Properties, or they can be stored as persistent objects.

The simplest storage representation is a text file. In the initialization of an object, text files are parsed to create run-time data structures. This representation is neither efficient and nor elegant. The second representation is a Java class called Properties. The Properties class represents a persistent set of properties. It is specialized for saving and retrieving data in the String class. This representation is faster than the first representation, because the Properties class is implemented with a hash table. However, the stored data must be in String. The third representation is a persistent object. The object can be an instance of a user defined class. Thus the storage can be an integer or a user defined table. Persistent object storage has the advantage that objects of any type can be stored and restored without conversion. Our project chooses persistent object storage, using Java Object Serialization [17].

After establishing persistent object storage as our choice, we need to figure out which objects to store. The main consideration is the object size and the abstraction model. Storing large objects may provide better abstractions and may be more convenient in the implementation. However, storing large objects may waste unnecessary disk space because only parts of the object contain the persistent data. Our system usually stores small objects, since the storing large objects does not necessarily provide better abstraction.

In general, this object storage model can be substituted with a more efficient model. Since the main goal of our project is not focused on database storage, the current object storage model is sufficient.

3.5 Summary

This chapter describes the design of the book paradigm system. After exploring the different possibilities in connecting client and server machine, the designs of the client and server is presented. The client is responsible for interacting with the user. The server is responsible presenting a book interface to a client, and organizing the underlying content of a book from a set of components.

Chapter 4

Implementation

The previous chapter discusses some high-level design issues on building the book paradigm system. This chapter presents the implementation of the system. The implementation utilizes Java applets and CORBA(Common Object Request Broker Architecture). The Java programming language is architecturally neutral and is portable across multiple machines. Building mobile code system is secure with Java, whose run-time system has built-in protection against viruses and tampering [8]. CORBA provides the mechanisms by which objects transparently make requests and receive responses through the CORBA ORB. [12] The CORBA ORB is similar to a "bus" on a hardware circuit [13]. It provides interoperability between applications built in different languages on different machines. The different langauges are mapped to a common interface called IDL(Interface definition language). By giving the ORB a remote object's uid(unique identifier), the ORB is able to return a reference to the object. In the prototype, uids are different from URLs(Uniform Resource Location). Run-time objects are distinguished by uids, while URLs represent the locations in persistent storage.

In this chapter, section 4.1 gives an overview of the different modules, and describes the interactions among the modules. Each other section focuses on a particular module and describes different algorithms in implementing the module.

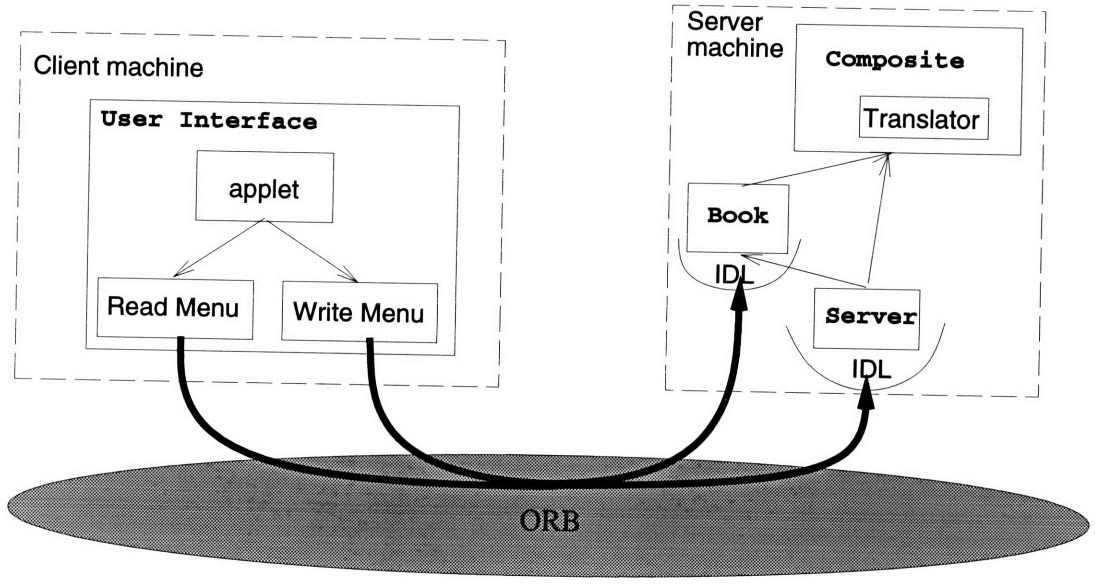


Figure 4-1: Implementation Overview.

4.1 Overview of Implementation

The implementation is divided into four main modules: the user interface module, the server module, the book module, and the composite module(See Figure 4-1). This section describes the basic functionality of each module, and the interactions among the modules.

The user interface module is responsible for all the interactions with a user. This module remotely invokes methods which are supported by the book module. The book module provides the basic methods for reading and writing, and supports the abstract structures of pages or chapters. For example, when a user wants to read a page, the requested page is saved as a temporary file in the server memory. Then the user interface module asks the associated web browser to display the content of the temporary file. The composite module maintains the content of a book. It combines a collection of files into a single virtual file, and provides the methods for reading and writing a selected region in bytes. While the book module presents a book interface, the composite module handles the underlying content.

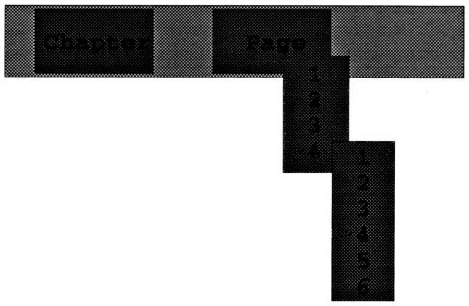


Figure 4-2: An example of a menu.

4.2 User Interface

The high-level design of the user interface was specified in section 2.3. This section discusses implementation details of the user interface. The user interface consists of three classes. The first class is an applet, which is loaded by the dynamic class loader on a web browser. The implementation of this class is fairly straight forward. The second class is a **read menu**. A **read menu** object is a window frame with an embedded menu. A menu has the items "Chapter" and "page". These items allow a user to request a desired chapter or page for reading. The third class is a **write menu**, which is a subclass of **read menu**. A **write menu** object allows a user to write over a desired chapter or page. This section focuses on the implementation of the two menu classes.

The two menu classes are responsible for all the interactions between the client and the server machine. At initialization, a menu searches for a **Server** object. It searches at the machine from which the applet originates. A **Server** IDL has the method **newBook**, which creates a new book and returns the book's *uid*. The menu invokes **newBook** remotely. It then asks the ORB to resolve the *uid* to obtain a reference to the new book. After the menu has obtained the reference, it gets the indices of the last chapter and the last page in the book. The menu needs the indices to generate the correct number of items. Since a page number may be very large, a chapter index or a page number is represented as a sequence of submenus as in Figure 4-2. For example, to access page 43, a user selects 4, and then selects 3.

The implementation of a menu is fairly tricky. In most submenus, there should be ten items, from zero to nine. However, some submenus contain less than ten items. For example, suppose a book has 46 pages. The "page" menu contains four submenu items from 1 to 4. Each of the 1, 2, and 3 submenus contains ten items. The 4 submenu contains 7 items. The problem can be represented by a tree of nodes(See Figure 4-3). This tree has two types of nodes. Type I nodes correspond to submenus such as 1, 2, and 3 in the example. A type I node has a branching factor of 10. Type II nodes correspond to submenus such as 4. A type II node has a branching factor of less than 10. To build this tree, a possible algorithm utilizes two recursions. Recursion A fills each menu with 10 submenus, and recursively calls recursion A at each submenus. Recursion B calls recursion A at type I nodes, and recursively calls Recursion B at type II nodes. The whole tree can be built by calling recursion B.

4.3 Server

Section 4.2 describes the interactions between a **menu** object and a **Server** object. This section describes the implementation of the **Server** class. The **Server** class supports two methods. The method **newBook** creates a new book instance at the server machine. It binds a *uid* with the new book instance, and returns the *uid*. The *uid* is generated by the **Server**. It distinguishes among different book copies at run-time. The method **main** instantiates the **Server** object, and **newBook** is the only method in a **Server** IDL. Figure 4-4 displays the creations of object instances.

The method **main** is implemented as follows.

1. Initialize the ORB which is in the Java IDL packages.

2. Instantiate a **Server** object and create a skeleton for the **Server** object. This skeleton acts as a proxy for the **Server** object.

3. Ask the ORB to bind the skeleton with the name of the **Server** object. Now a client can ask the ORB to resolve the **Server** object with its name.

The **newBook** method is implemented as follows. The input parameters of **newBook** are *url* and the *name* of the book. The *url* specifies the location of the book.

It is different from the *uid*, which is generated by the Server to distinguish among different book copies at run-time..

1. Instantiate a **Composite** object using *url* as an input. The *url* input describes the URL of the **Composite** object.
2. Create a *uid* for the **Book** instance. This *uid* will be useful for identifying a unique path which stores data for this book.
3. Instantiate a **Book**. Two of the inputs are the **Composite** object instantiated in Step 1 and the *uid* created in Step 2. **Book** is an abstraction layer which wraps around the **Composite** object and presents the interface of a book.
4. Create a **skeleton** for the book instance. This skeleton acts as a proxy for the book instance. Ask the ORB to bind the skeleton to the *uid* created in 2.
5. Return *uid*. A client can obtain a reference to this book instance with the *uid*.

4.4 Book

A client applet can remotely invoke methods on a **Book** object. A **Book** IDL specifies the common interface between the client applet and the **Book** class. This IDL is analogous to a *role* in the Information Mesh. A **Book** class implements the methods on the IDL. Section 4.4.1 will present the methods in the **Book** IDL, and the rest of section 4.4 will describe the implementation of the **Book** class.

4.4.1 Book's Interface

The **Book** IDL is analogous to the book role interface of the Information Mesh Object Model. The IDL consists of the following methods:

- **maxPage()** Return the maximum page index.
- **maxChapter()** Return the maximum chapter index.

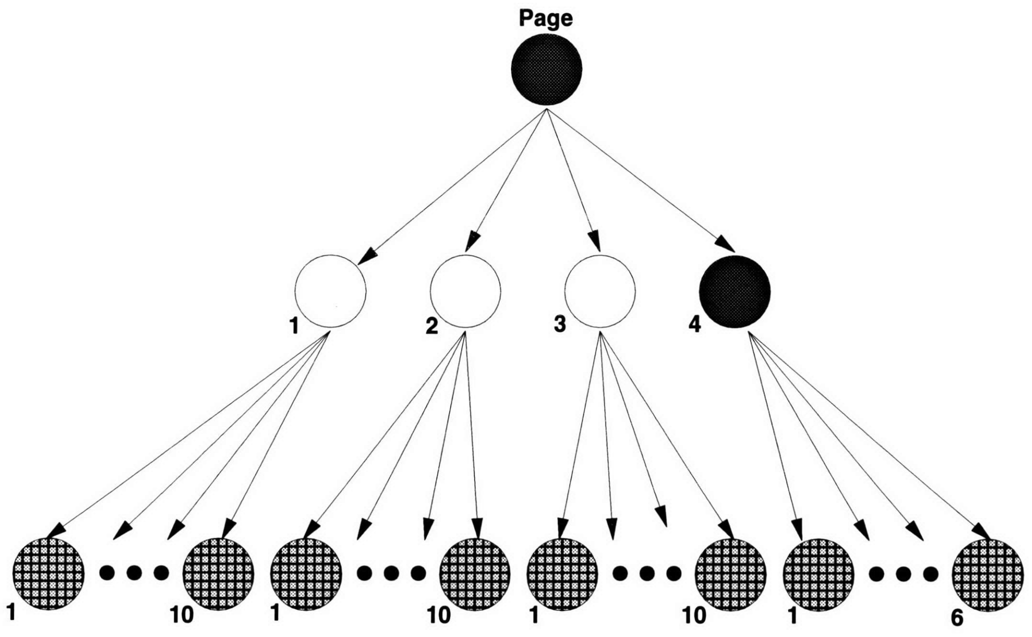


Figure 4-3: A tree of nodes.

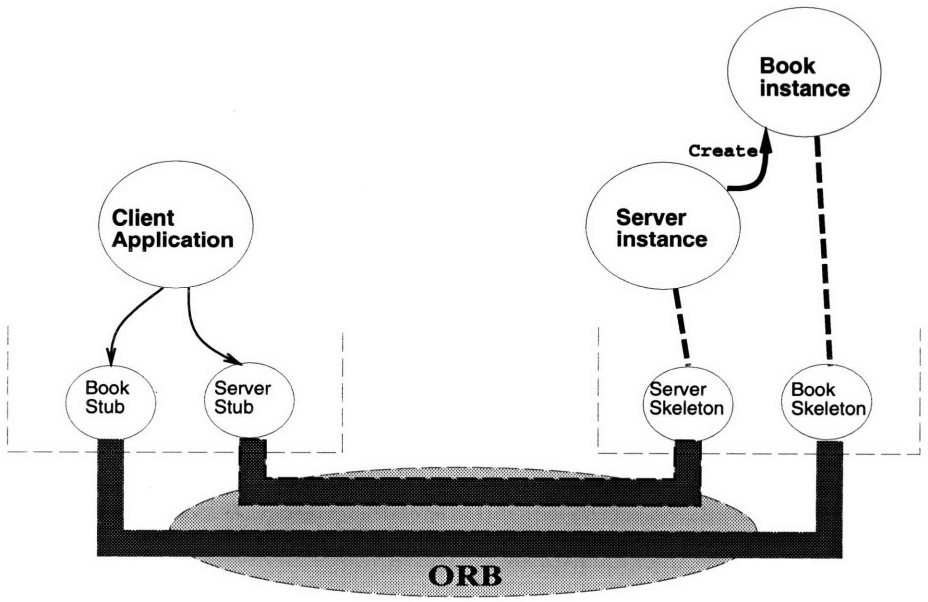


Figure 4-4: Object instance creation.

- **pageSize(int)** Return the size of a specified page.
- **page(int)** Save the specified page in a temporary file. Return the URL of the temporary file.
- **chapter(int)** Save the specified chapter in a temporary file. Return the URL of the temporary file.
- **setPageSize(int, int)** Modify the size of a specified page.
- **replacePage(string, long, int)** Replace the specified page with a new file. The string parameters indicate the new file's URL. The other parameters indicate the new file's size and the index of the replaced page.
- **replaceChapter(string, long, int)** Replace the specified chapter with a new file. The string parameters indicate the new file's URL. The other parameters indicate the new file's size and the index of the replaced chapter.

4.4.2 Book Implementation

A **Book** object holds references to the following objects: a **Composite** object, a list of entries which point to the beginnings of chapters, and a list of entries which point to the beginnings of pages(See Figure 4-5). The **Composite** object represents the underlying content of a book. The content is accessed by bytes. The offset from the beginning of a composite is indicated by a **Filepointer** in bytes. The **Book** module allows clients to access the content by chapter or by page. The list of chapter pointers maintains the exact location of every chapter. Similarly, the list of page pointers maintains the exact location of every page. Every location is specified by a filepointer in the **Composite** object.

When a book is initialized, a temporary directory is created for this book. The directory will store all temporary files which represent chapters and pages of this book. At the same time, the list of chapter pointers and the list of page pointers are restored from persistent storage. In persistent storage, a list may exist in one of two

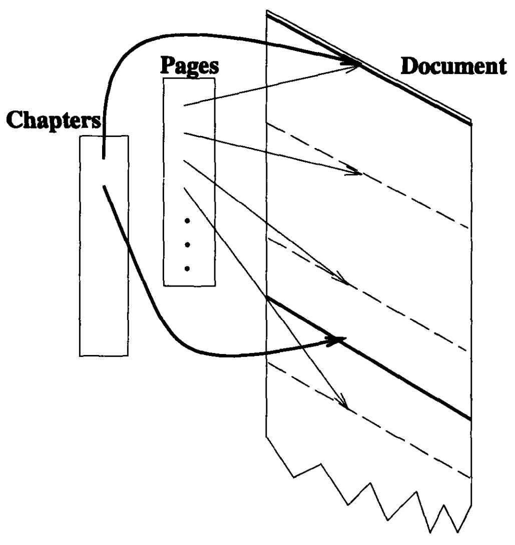


Figure 4-5: The lists of pointers.

possible forms. It may be represented by a text file, or it may exist as a persistent object. If a list is represented by a text file, the text file is parsed during initialization in order to restore the list object. If a list exists as an object in persistent storage, it can be restored directly into an object.

4.4.3 Reading Pages and Chapters

When a user decides to read a page or a chapter, the **menu** remotely invokes the methods **page(int)** and **chapter(int)** on a book. The implementations for **page(int)** and **chapter(int)** are very similar. This section describes the implementation of **page(int)** and the differences between **page(int)** and **chapter(int)**.

page(int)

1. Choose a path for a temporary file.

2. Invoke **Composite.readPT**. **Composite.readPT** reads the content between a starting filepointer and an ending filepointer, and places the content into a temporary file. The filepointers are retrieved from the list of page pointers. The starting filepointer is at the requested page index. The ending filepointer is at the subsequent index. The path of the temporary file is the path created in

Step 1.

3. Add a begin HTML tag to the top of the temporary page file, and add an end HTML tag to the bottom of the temporary page file. This step ensures that a web browser reads the file in HTML format. Note that it is not possible to add these tags inside a **Composite** object, because chapters and pages may overlap.

4. Update the hashtable which keeps track of which page file has been produced. When this page is requested again, steps 1 to 4 will be skipped.

5. Return the path for the temporary page file.

Note that Step 4 may generate an inconsistency between the copy which is read and the actual copy of the book. For example, suppose someone rewrites a page which is simultaneously read by another person. Then the temporary page file is consistent with the new page. However, the book paradigm does not specify a locking mechanism. Thus the inconsistency issue will not be considered in this thesis.

4.4.4 Replacing Pages and Chapters

The **Book** module also supports the methods **replacePage** and **replaceChapter**. These two methods modify a book's content. When **replacePage** is invoked, a page is replaced with the content in a new file. If the size of the new file is larger than the size of the replaced page, the content in the file may represent several new pages. If the file size is less than the page size, the content in the file may represent a segment of a page. When **replaceChapter** is invoked, a chapter is replaced by the content in a new file. The new file is a new chapter. The implementation of the **replacePage** and **replaceChapter** are more complicated than **page** and **chapter**. This section first describes **replacePage**, and then compares this method with **replaceChapter**.

The method **replacePage** takes in three input parameters: the *url* of the new file, the size of this new file, and the index of the replaced page. The implementation of the **replacePage** is outlined as follows(See Figure 4-6):

1. Create a new directory. This directory stores the new components of the book. The name of the directory is the current book's name concatenated with the book's *uid*. If this directory already exists, then no new directory is created.

2. Transport the new file from its *url* to the new directory created in Step 1. This step is necessary since the new file may only exist temporarily. For example, this file may be generated when someone edits a page. The person may delete the file after calling **replacePage**.

3. Invoke **Composite.writePT**. The inputs are (1) the filepointer which indicates where to start replacing, and (2) the filepointer which indicates where to stop replacing. (3) the path of the new file transported in Step 2.

4. Call the private procedures **updatePgLink**, **updateChpLink**, and **shiftChpLinkFpt**.

   (a) **updatePgLink** updates the list of page pointers. If the new page is much larger than the old page, the procedure appends extra page pointers to the end of the list. If the new page is much smaller than the old page, the procedure may delete some existing page pointers.

   (b) **updateChpLink** modifies the list of chapter pointers. The procedure deletes chapter pointers which correspond to chapters on the replaced page. The procedure also inserts chapter pointers which correspond to extra chapters on the new file.

   (c) **shiftChpLinkFpt** shifts the chapter pointers which correspond to pages after the replaced page. The shifting is necessary when the size of the replaced page is different from size of the new file.

The method **replaceChapter** is almost identical to **replacePage**. There are only two differences. First, **replaceChapter** uses the chapter pointers instead of page pointers when it invokes **Composite.writePT**. Second, **replaceChapter** does not call **updateChpLink**, though it does call **updatePgLink** and **shiftChpLinkFpt**.

Updating the List of Page Pointers

The method **updatePgLink** updates the list of page pointers. The input to **updatePgLink** is **oldLength**, which is the length of the composite object before it was modified. The implementation is outlined as follows:

1. Retrieve the length of the composite object. Call it **newLength**.
2. Compute the average size of the first ten pages.
3. If **newLength** is greater than **oldLength** by more than the average size computed in Step 2, then append page pointers to the end of the list of page pointers. To append new page pointers to the end of the list:
    (a) Delete the last page pointer, which points to the end of the composite object. The value equals the size of the composite object.
    (b) Append new page pointers to the end of the list. Each new page pointer is larger than the previous one by average page size computed in Step 2. Stop before the new page pointer is greater than **newLength**. Add the last page pointer whose filepointer is **newLength**.
4. If **newLength** is less than **oldLength**, then delete page pointers from the end of the list. Stop deleting when the last page pointer is less than **newLength**. Then add the last page pointer whose filepointer is **newLength**.

Updating the List of Chapter Pointers

The method **updateChpLink** updates the list of page pointers. The input to **updateChpLink** are (1) the index of the page to be replaced, and (2) the URL of the new file which will replace the specified page. The implementation is outlined as follows:

1. Find the index of the first chapter which appears on the page to be replaced. Call this index **first index**. Find the index of the first chapter which appears AFTER the page to be replaced. Call this index **subsequent index**.

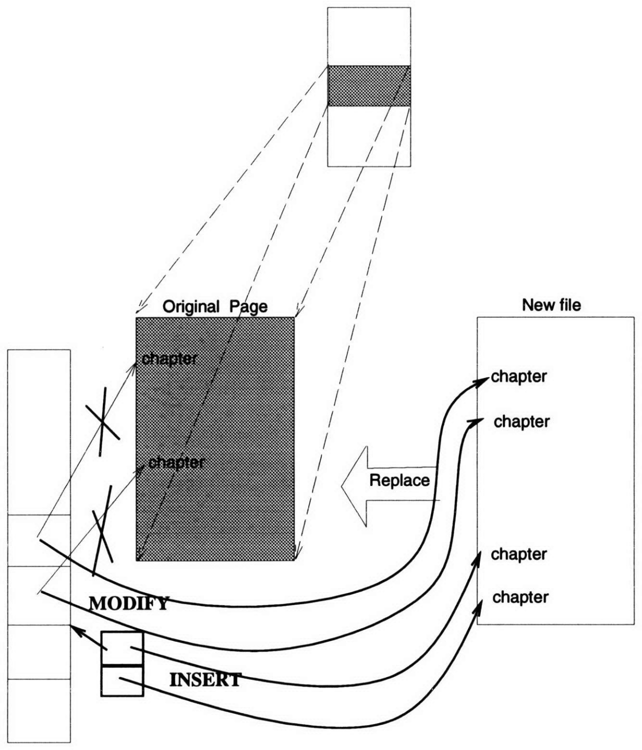


Figure 4-6: Replaceing a page.

(a) Retrieve the filepointer which points to the beginning of the page to be replaced. Call this **Filepointer A**. Retrieve the filepointer which points to the beginning of the next page. Call this **Filepointer B**.

(b) Let the current chapter pointer be the first chapter pointer. Compare the current chapter pointer with the filepointers retrieved in Step 1(a).

   i. If the chapter pointer is greater than **Filepointer B**, set the current index to be **subsequent index**. Then go to Step 2.

   ii. If the chapter pointer is greater than **Filepointer A**, set the current index to be **first index**. Then go to Step 1(c).

   iii. Otherwise, repeat Step 2 with the next chapter pointer as the current chapter pointer.

(c) Starting from chapter pointer at **first index**, iteratively check whether the current chapter pointer is greater than **Filepointer B**. If the current chapter pointer is greater, set the current index to be **subsequent index**, and break out of the iteration.

2. Remove any extra chapter pointers which point at the replaced page, and update other chapter pointers. For example, if two chapters point to the replaced page and one chapter points to the new file, the second chapter pointer is removed, and the first chapter pointer is updated to the correct location.

   Let the current index equal **first index**. Until the current index is less than **subsequent index**, do the following for each index:

   (a) Invoke **findOffset** to find the filepointer of the chapter which corresponds to the current index.

       findOffset parses the new file, searching for the special tag which represents a chapter. It returns the offset from the beginning of the file. This offset will be one of the inputs when **findOffset** is called again. So the next call on **findOffset** will yield the offset for the next chapter.

   (b) If **findOffset** finds a chapter on the new file, the current chapter pointer is set to the **Filepointer A** + offset.

   (c) If there is no additional chapter on the new file, remove the entry of the current index. Since an entry has been removed, decrement **subsequent index**.

3. Insert additional chapter pointers which correspond to any additional chapters in the new file. For example, suppose Chapter 1 was on the original page and three chapters are on the new file. Step 2 updates the first chapter pointers, and this step inserts two more chapter pointers.

4.4.5 Other methods

The IDL interface also provides the methods **maxChapter()**, **maxPage()**, **pageSize(int)**, and **setPageSize(int, int)**. The implementation for each of these methods are fairly straight forward.

- **maxChapter()** returns the highest index in the list of chapter pointers.

- **maxPage()** returns the second highest index in the list of page pointers. Note that the second highest index represents the maximum page index whereas the highest index points to the end of the document.

- **pageSize(int)** computes the difference between the filepointer at the requested index and the filepointer at the next index. Then it returns the difference.

- **setPageSize(int, int)** (1) shifts the filepointers with indices higher than the requested index(except the highest index) (2) removes all indices whose filepointers are greater than the size of the composite object.

4.5 Composite

An instance of the **Composite** class behaves like a single, virtual file while its content is composed of a number of disk files. Each region of a composite object can be specified by a starting filepointer and an ending filepointer. A filepointer represents the offset, in bytes, from the beginning of a composite. Since these filepointers point to a *virtual* file, they will be called *virtual* filepointers for the rest of this report. A virtual filepointer is analogous to a real filepointer, which represents the offset, in bytes, from the beginning of a disk file.

A disk file cannot be accessed directly by a virtual filepointer. In order to read or write disk files, virtual filepointers must be translated into their corresponding filepointers of disk files(See Figure 4-7). Each composite object contains an instance of the **Translator** class. A translator is responsible for translating virtual filepointers into their corresponding filepointers of disk files. Each translation takes a virtual filepointer as an input, and returns its corresponding filepointer and the URL of its corresponding disk file.

A composite object may have several levels of components. For example, a composite may contain a component which in turn contains another component. In our implementation, each composite only needs to deal with one level of components. It sees its components as composite objects which are able to deal with their own com-

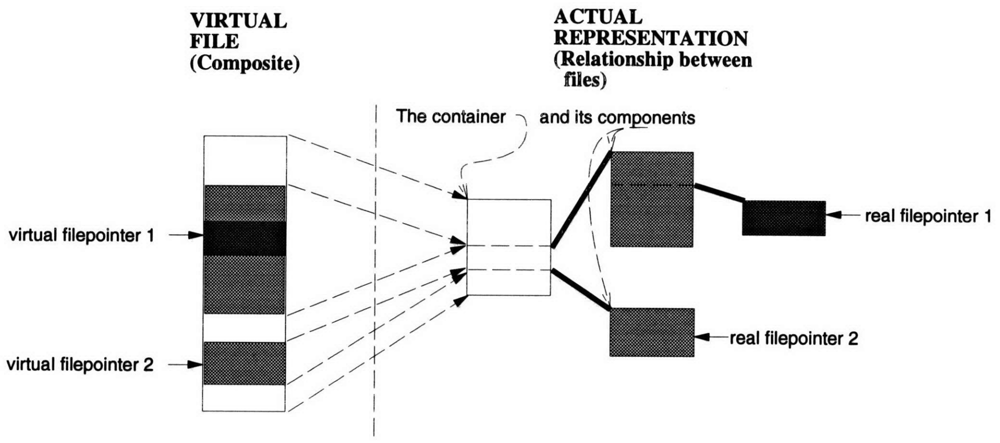


Figure 4-7: Mapping between a composite object and its underlying file structure.

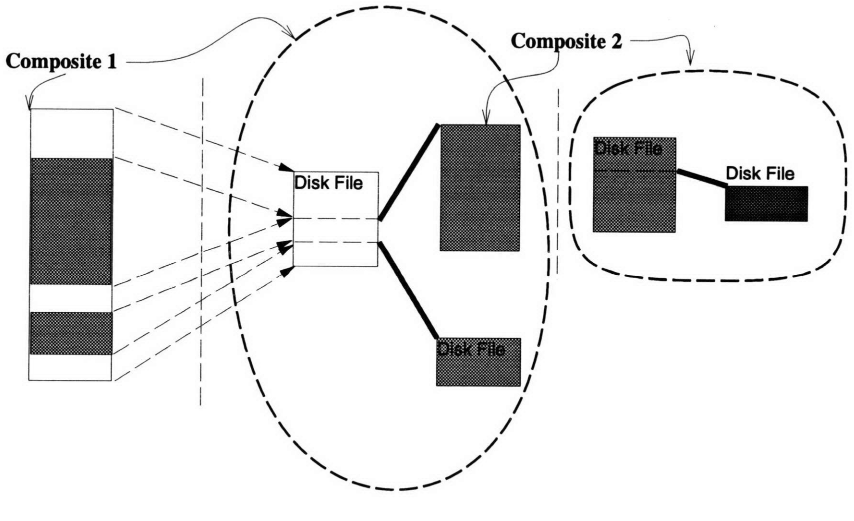


Figure 4-8: The Recursive nature of a composite object.

ponents(see Figure 4-8). Inside each composite object, a disk file holds the references to the components. This disk file will be called a **container** for the rest of this report. A composite object always has one container and it may have several components.

By restricting composites to handle one level of components, the implementation becomes simpler. For example, a translator translates a virtual filepointer in a composite into the corresponding filepointer in a component or container. The filepointer in a component may be virtual or real. If the component is a composite, than the filepointer is virtual. If the component is a disk file, than the filepointer is real.

The rest of section 4.5 describes the implementation of the **Translator** class and the implementation of the **Composite** class respectively. Section 4.5.1 describes the implementation of the **Translator** class. Section 4.5.2 describes the **Composite** class and the algorithms for reading and writing a composite object.

4.5.1 Translator

The Translator class is responsible for translating a virtual filepointer in a composite into the corresponding filepointer in a component or container. Inside a translator, a composite object is partitioned into different regions. Each partition is either a component or a segment in the container between two insertions of components. A translator describes its partitions by the **partition array**. Each array entry points to the beginning of each partition. The array begins from index=0. A composite object is allowed to read and modify the **partition array** inside its translator.

At initialization, a **partition array** is created by reading data from a **table of components**(See Figure 4-9). A **table of components** stores the data which describe the relationships among a container and its component. The composite object is allowed to read and modify the **table of components** inside its translator. The table is implemented as a list of component entries. Each component entry keeps the following data on the the corresponding component:

- The URL of this component.
- The offset(filepointer) to insert this component into the container.

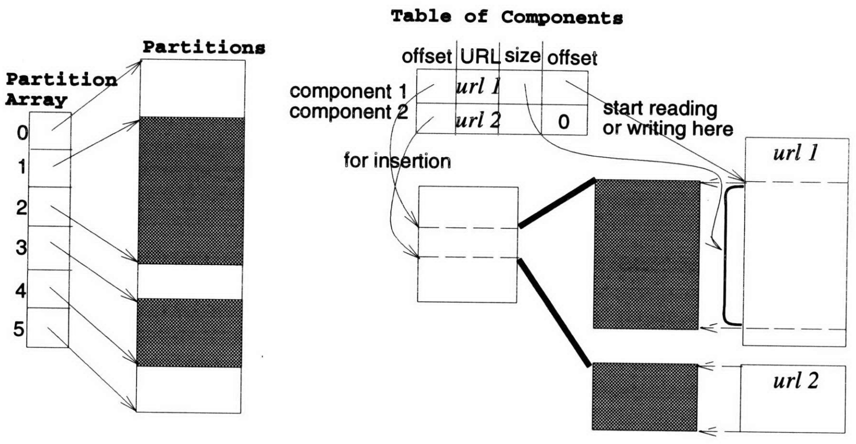


Figure 4-9: Partition Array and Table of Components.

- The size of this component.
- The offset on this component to begin reading or writing.

Both a **partition array** and a **table of components** describe the relationships among a container and its components. A **partition array** divides a composite into partitions, and keeps track of which of the partitions are components. A **table of components** maintains the relationships between a container and its components.

Translation

The method **getRealFilepointer** translates a virtual filepointer in a composite into a filepointer in a component. The filepointer in a component is represented by an array of two entries. The first entry is the URL of the component. The second entry is the filepointer. The algorithm for **getRealFilepointer** is as follows.

1. One of the input parameters is the virtual filepointer which will be translated. Compute the index of the partition which contains this input filepointer. The index is computed by comparing the input filepointer with **partition array** entries, using binary search.

2. From the **partition array**, retrieve the filepointer in the entry whose index equals to the index computed in Step 1. Compute the difference between this filepointer and the input filepointer. Let **offset** be the difference.

3. If the partition in Step 1 is a component, find the *url* of the component from the **table of components**. Return an array of *url* and **offset**.

4. If the partition in Step 1 is a segment of the container, then, from the **table of components**, find the component entry which corresponds to the previous partition. Add **offset** to the insertion filepointer of this component entry. The result is the real filepointer in a disk file. Return an array of string "this" and the result. The string "this" indicates the partition is a segment of the container.

Initialization

A **partition array** is initialized from a **table of components**, which is retrieved from persistent storage. The initialization algorithm is correct when the following assumptions hold: (1) an even index partition corresponds to a region in the container; (2) an odd index partition corresponds to a component. Then, the following formula computes the beginning filepointer for each partition, using the data in the **table of components**.

Let

b = *The virtual filepointer of the beginning of a partition.*

l = *The filepointer for inserting a component into a container.*

s = *The size of a component.*

i = *The index of a partition.*

$$b_i = l_{\lfloor \frac{i-2}{2} \rfloor} + \sum_{j=0}^{\lfloor \frac{i-2}{2} \rfloor} s_j$$

Intuitively, a virtual filepointer corresponds to the result from shifting a real filepointer by the cumulative size of all previous components. The cumulative size of all previous components is represented by $\sum_{j=0}^{\lfloor \frac{i-2}{2} \rfloor} s_j$. Since l is the filepointer for inserting

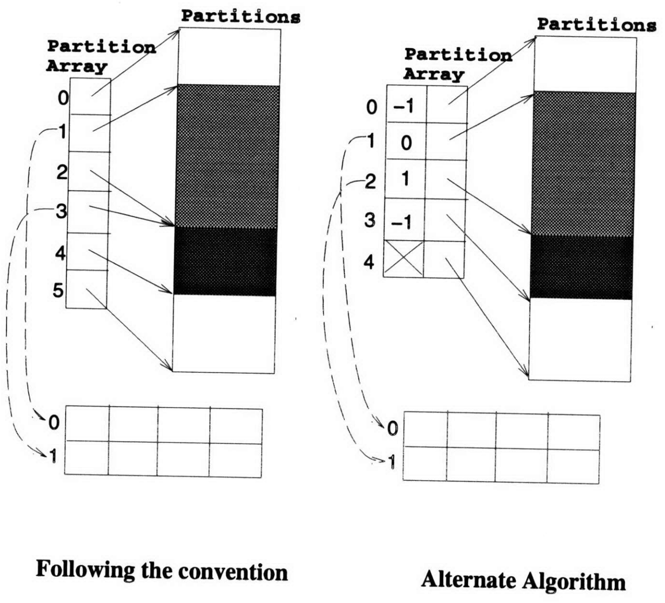


Figure 4-10: Convention, and alternate algorithm.

a component into a container, shifting l by $\sum_{j=0}^{\lfloor\frac{i-2}{2}\rfloor} s_j$ yields the virtual filepointer of the beginning of a partition.

Implications of the assumption

The algorithm for initialization is correct assuming that the partitions with odd indices correspond to segments of the container, and the partitions with even indices correspond to the components. However, this assumption does not have to hold. For example, a container may insert two components at the same filepointer. Then one of the components corresponds to a partition with an odd index. To keep the assumption valid, the following convention is introduced. Whenever a container inserts two components at the same filepointer, insert a partition of zero size between the two components(See Figure 4-10). This partition of zero size is only seen in the **partition array**.

By introducing this convention, the algorithm for initialization works properly. On the other hand, introducing this convention has the following disadvantages. First, it increases the overall complexity in the **partition array**. For example, consider a virtual filepointer which points to the beginning of a partition. This partition corresponds to a component, and its preceding partition is a partition of zero size. When a translator computes which partition contains the filepointer, it must make sure that the result is not the partition of zero size. Second, after writing the composite a number times, partitions of zero size may appear often more than necessary. These unnecessary partitions need to be cleaned up.

Alternate algorithms may be better and simpler. For example, instead of using even-odd parity to determine whether a partition is a component, a new field is added to each entry in the **partition array**. The new field indicates whether the partition is a component or not. If the partition is not a component, the value of the field is -1. Otherwise, the value of the field is the number of previous components. Thus, each **partition array** entry has a direct reference to the corresponding component entry in the **table of components**. However, this alternative also has a drawback because when a component is inserted in the middle, the fields for all the suceeding components must be incremented. Since the system was not built this way, this alternative may have other problems yet to be discovered.

4.5.2 The Composite Class

The Composite class has an interface similar to a disk file. The class supports the methods **readPT** and **writePT**. The method **readPT** reads from a specified region in a composite object. The method **writePT** writes over a specified region in a composite object. A specified region is bounded by a starting virtual filepointer and an ending virtual filepointer. The algorithms for **readPT** and **writePT** are presented as follows.

Algorithm for readPT

In the algorithm for **readPT**, the basic idea is to fill a temporary file with individual "chunks" of text. Each "chunk" of text may come from a full partition or a segment of a partition.

1. Translate the starting and ending filepointers into real filepointers by invoking **Translator.getFilepointer**. Let the real filepointers be **startoffset** and **endoffset** respectively.

2. If the starting virtual filepointer and the ending virtual filepointer belong to the same partition, then transfer the content of the partition into a temporary file. The transfer starts at **startoffset** and stops at **endoffset**.

3. If the starting virtual filepointer and the ending virtual filepointer belong to two distinct partitions:

   (a) Transfer the content of the first partition into a temporary file. The transfer starts at startoffset and stops at the end of the partition.

   (b) For each subsequent partition which does not contain the ending virtual filepointer, concatenate the temporary file in 3(a) with the content of the entire partition. The partition may be a component or a segment of the container.

   (c) Transfer the content of the last partition into a temporary file. The transfer starts at beginning of the partition and stops at **endoffset**.

4. Return the *url* of the temporary file.

To transfer the content of a partition into the temporary file, the algorithm splits into two cases. In the first case, the partition corresponds to a segment of the container. In this case, the container is a disk file. Therefore the algorithm reads from the disk file and appends the content into the temporary file. In the second case, the partition corresponds to a component. The component may be a composite object, and it may contain another level of components. Therefore the algorithm recursively invokes **readPT** on its component. The basic steps are:

1. Instantiate a composite object which is the component.

2. Invoke **readPT** on the composite object instantiated in Step 1.

Algorithm for writePT

There are two general schemes for implementing the method **writePT**. The first scheme restricts a composite object to write only files in the local memory. The second scheme allows a composite object to write files in the local memory and files in a remote memory. The following discusses the two schemes, their relative advantages and disadvantages, and the implementation of one of the schemes.

Scheme 1 :

The first scheme restricts a composite to write only files in the local memory. In this scheme, the abstraction is analogous to "pasting" new components on top of a composite. The new components are new files in the local memory. After the "pasting", the partition boundaries of a composite are changed. For example, suppose a component is pasted on top of a composite which contains no component. Thus the original composite has only one partition. The modified composite has three partitions. The middle partition corresponds to the new component(See Figure 4-11). The basic steps in carrying out this scheme are as follows.

1. At the local machine, create a new file which contains the content of the new component.
2. The new component may be "pasted" on top of certain regions of the container. Cut out this region from the container.
3. Update the **partition array** and the **table of components** to be consistent with the following circumstances:
   (a) A composite does not contain those components whose partitions have been completely covered by the new component. For example, if a component is pasted on top of several partitions, the component may completely cover certain partition.
   (b) Some partitions are partially covered by the new component. If such a partition corresponds to a component, a composite cannot delete the

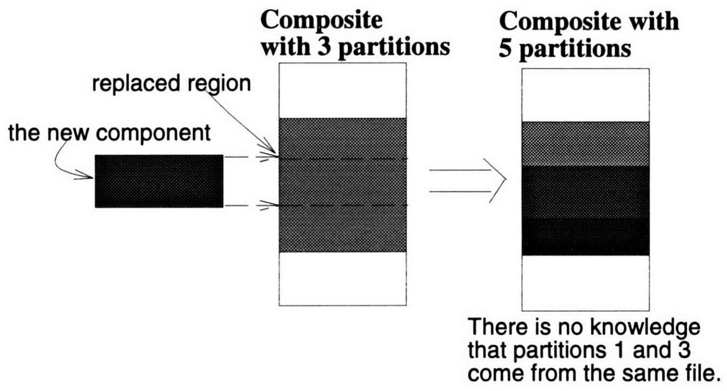

Composite with 3 partitions
Composite with 5 partitions
replaced region
the new component
There is no knowledge that partitions 1 and 3 come from the same file.

Figure 4-11: Scheme 1.

covered area. Thus a composite may contain only a segment of a disk file(or a composite) as its component. A composite needs to maintain the information which specifies the boundary of the segment. The information is maintained in the **table of components**.

In many situations, not all of the component files can be modified. Writes are restricted for certain component files. Scheme 1 is useful in this type of situation. It creates a private version of a new composite in the local memory. For example, in a collaborative environment, editors may keep their own versions of a composite before updating the different versions to the central copy. On the other hand, this scheme has several drawbacks. First, the design for garbage collection is not well specified, since writing component files is restricted. Second, most of the component files will reside in the local memory after writing the composite a number of times. Third, a high degree of complexity may be introduced when the **partition array** and the **table of components** are updated. Our book paradigm system is built with this scheme, and the high degree of complexity is evident in the implementation of the system.

Scheme 2 :

The second scheme allows a composite object to write files in the local memory and also files in a remote memory. In this scheme, the method **writePT** utilizes a recursive algorithm, similar to the algorithm of **readPT**. During each recursion, it modifies the content within the container, and invokes the method **writePT** on its component(See Figure 4-12). The basic steps in carrying out this scheme are as follows.

1. Set *current partition* to be the partition which contains the starting filepointer.
2. If the *current partition* corresponds to the container, then
   (a) If the partition contains the ending filepointer, replace the region between the starting filepointer and the ending filepointer with the remainder of the new file.
   (b) Otherwise, replace the content between the starting filepointer and the end of the *current partition* with a segment of the new file. The segment has the same size as the replaced region
3. If the *current partition* corresponds to a component, instantiate a composite object which represents the component. Invoke **writePT** on the composite object.
4. Set the *current partition* to be the next partition. Repeat from Step 2, replacing starting filepointer with start of the *current partition.*
5. Break out from steps 2 to 4 when the new file has reached the end. Delete the region between the ending filepointer and the filepointer where replacement from the new file stops.

Scheme 2 has the following advantages. First, scheme 2 is less complex than scheme 1. The lower complexity is evident in the manipulation of the **partition array**. Under scheme 2, the **partition array** does not change much after the invocation of **writePT**. Second, after writing the composite a number of times, components are still distributed over different sites' memory . In scheme 1, most components will reside in the local memory after many writes. Third, a possible advantage may come in the performance. Each invocation of **writePT** on a remote component may be taken up by a distinct thread. Then

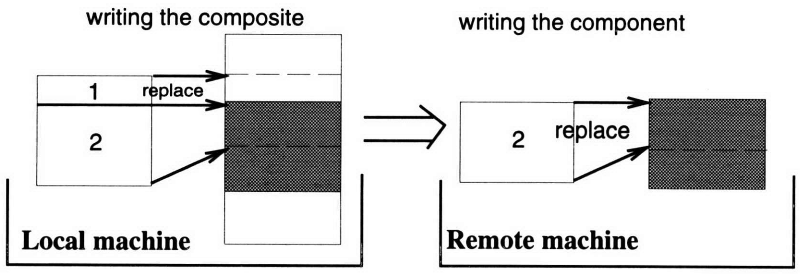


Figure 4-12: Scheme 2.

the **writePT** process in the local machine may proceed while writing components are computed in remote machines. The disadvantage of scheme 2 is that certain components may be restricted from writing. For example, security reasons may restrict the modification of certain components. Then scheme 2 may become complex.

Implementation of Scheme 1

Our book paradigm system is built using scheme 1. The implementation of scheme 1 is divided into two phases. Figure 4-13 shows the control flow of scheme 1. The first phase deletes the partitions which will be completely covered by the new component. This phase first deletes the corresponding entries in the **partition array**, and cuts out a region from the container. The cut-out region is the area covered by the new component.

The second phase constructs a relationship between the new component and the container. This phase modifies the **partition array** and the **table of components** so that the composite recognizes the new component. After the first phase cuts out the covered region, the second phase is left with four cases to consider(See Figure 4-14). The cases are distinguished by the starting and ending input filepointers. An input filepointer is a virtual filepointer on the composite object. After it is translated into a real filepointer, it may point to either the container or a component. In case 1, both the starting filepointer and the ending filepointer point to a component. In case 2, the starting filepointer points to a component, and the ending filepointer points

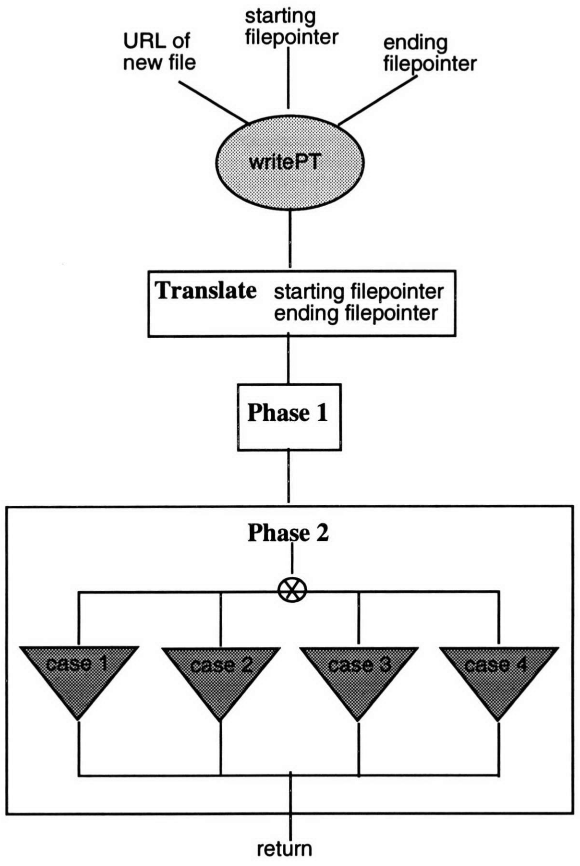


Figure 4-13: The control flow of the method writePT.

to the container. In case 3, the starting filepointer points to the container and the ending filepointer points to a component. In case 4, both filepointers point to the container. In each case, the **partition array** and the **table of components** are handled differently. The implementations of phase 1 and phase 2 are outlined as follows.

Phase 1 :

Two of the input parameters are the starting virtual filepointer, **startpt** and the ending filepointer, **endpt**. These two filepointers indicate the region to be replaced in the composite object.

1. Find the partitions to which **startpt** and **endpt** belong. Call them **starting partition** and **ending partition**.

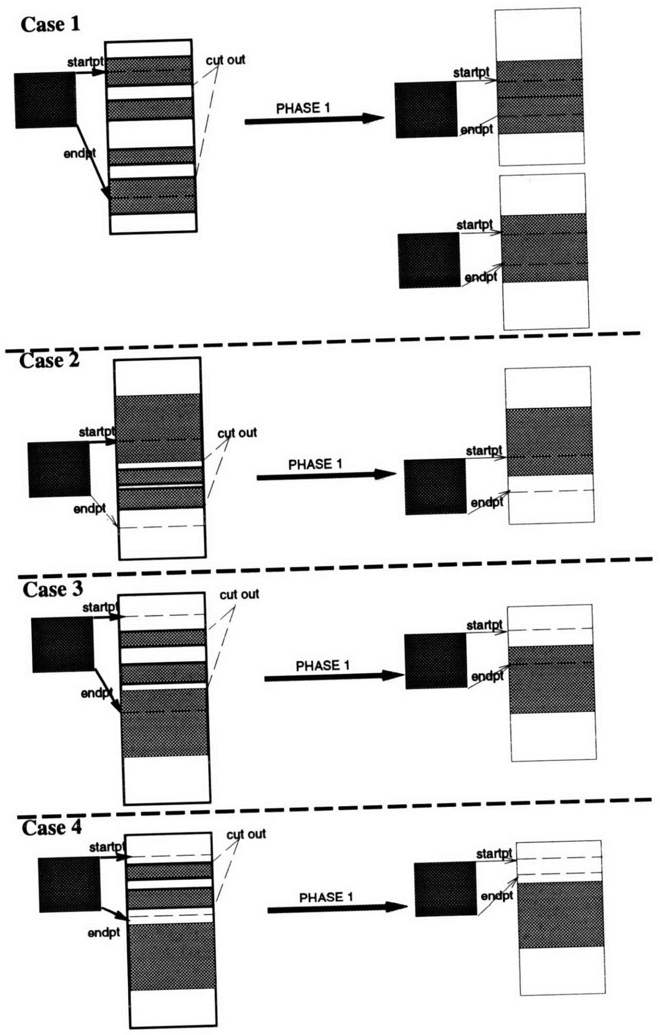


Figure 4-14: Reducing to four cases.

2. If the starting partition corresponds to a component, find the index of this component in the **table of components**. If the starting partition corresponds to the container, find the index of the component which corresponds to the next partition. Call this index **start component index**.

   If the ending partition corresponds to a component, find the index of this component in the **table of components**. If the ending partition corresponds to a container, find the index of the component which corresponds to the previous partition. Call this index **end component index**.

3. From the container file, cut out the region which is covered by the new component. This region is bounded by two filepointers. The first filepointer indicates where to insert the component of the **start component index**. The second filepointer indicates where to insert the component of the **end component index**.

4. Remove the entries between **start component index** (exclusive) and **component index** (exclusive) from the **table of components**.

5. Remove the entries which point to partitions between **starting partition** (exclusive) and **ending partition** (exclusive) from the **partition array**.

6. For each index after **start component index**(inclusive), shift the filepointers which indicates the offset to insert the component. These filepointers are shifted by the size of the cut-out region in Step 3.

   For each partition after **ending partition**(inclusive), shift the corresponding **partition array** entry by the $startpt - endpt$.

Phase 2 :

Phase 2 is branched into four cases. In the following description, each case focuses on the three or four partitions where most changes occur. For each case, *delta* is defined to be the difference between the new document size(*filesize*) and the size of the replaced region. Thus $delta = filesize + startpt - endpt$.

Case 1:

1. If **startpt** and **endpt** points to the same partition,

   (a) Insert two entries of **startpt** and two entries of **endpt** into the **partition array**. Thus two zero length partitions are created. These two zero length partitions deal with the even and odd requirement, so that even indices always correspond to components.

   (b) Shift all entries by *delta*, starting from the two entries whose filepointers equal **endpt**.

   (c) Modify the **table of components**. The original **table of components** has one component entry. The new **table of components** has three component entries. The first entry corresponds to a top segment the original component. The top segment ends at the insertion of the new component. The third entry corresponds to the remainder of the original component. The second component entry corresponds to the new component. The fields of the **table of components** are updated as shown in Figure 4-15.

2. Otherwise, the **starting partition** and the **ending partition** are beside each other,

   (a) Do steps 1(a) and 1(b).

   (b) Delete the entry of the **ending partition** from the **partition array**. This entry is no longer useful, because the new component covers the boundary between the **starting partition** and the **ending partition**.

   (c) Modify the **table of components**. The original **table of components** has two component entries. The new **table of components** has three component entries. The first entry corresponds to the first original component. The third entry corresponds to the second original component. The second component entry corresponds to the new component. The fields of the **table of components** are updated as shown in Figure 4-16.

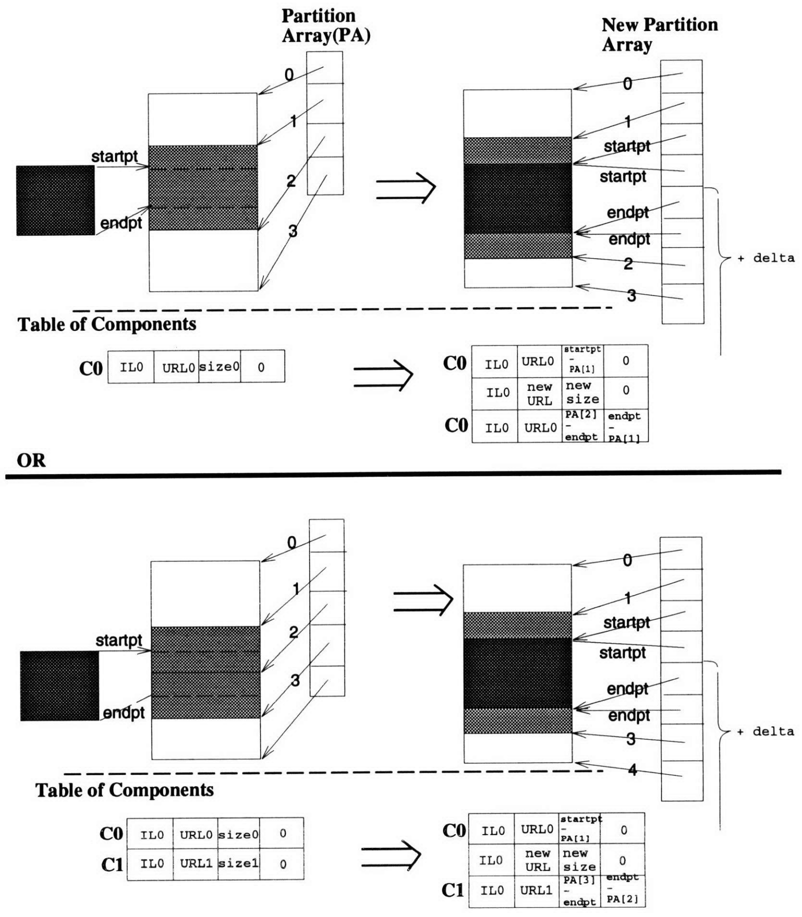


Figure 4-15: Case 1.

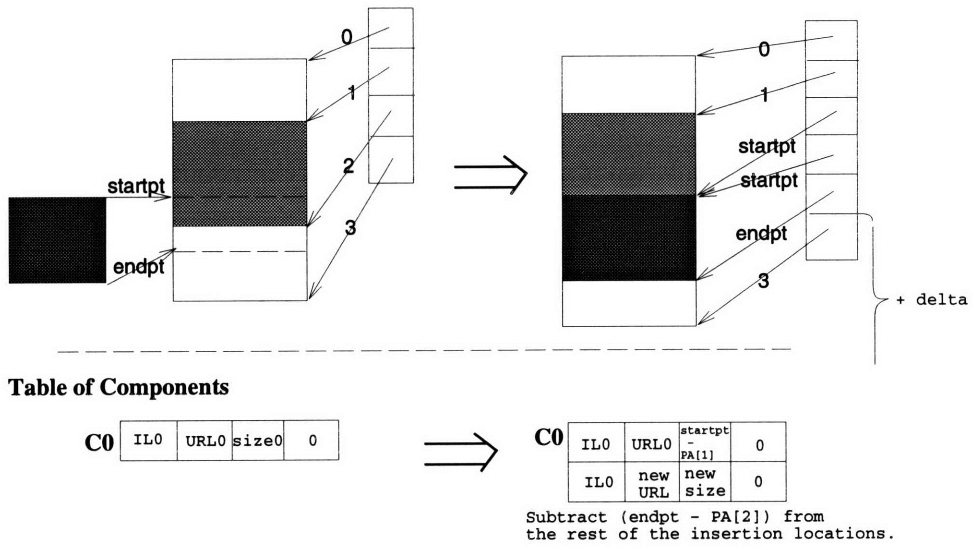


Figure 4-16: Case 2.

Case 2:

1. Insert two entries of **startpt** and one entry of **endpt**.
2. Delete the entry of the **ending partition** from the **partition array**.
3. Shift all entries by *delta*, starting from the entry whose filepointer equals **endpt**.
4. Modify the **table of components**. The original **table of components** has one component entry. Call the component of this entry the "original" component. The new **table of components** has two component entries. The second component entry corresponds to the new component. The fields of the **table of components** are updated as shown in Figure 4-13.
5. Cut out a region from the container. The region starts from the insertion of the "original" component. It ends at the real filepointer which corresponds to **endpt**.
6. For all subsequent component entries, shift their insertion filepointers by the size of the cut-out region in Step 5.

Case 3:

1. Insert one entry of **startpt** and two entries of **endpt**.
2. Delete the entry of the **starting partition** from the **partition array**.
3. Shift all entries by *delta*, starting from the two entries whose filepointers equal to **endpt**.
4. Modify the **table of components**. The original **table of components** has one component entry. Call the component of this entry the "original" component. The new **table of components** has two component entries. The first component entry corresponds to the new component. The fields of the **table of components** are updated as shown in Figure 4-17.
5. Cut out a region from the container. The region starts from the real filepointer which corresponds to the **startpt**. It ends at the insertion of the "original" component.
6. For all subsequent component entries(including the entry of the "original" component), shift their insertion filepointers by the size of the cut-out region in Step 5.

Case 4:

1. Insert one entry of **startpt** and one entry of **endpt**.
2. Shift all entries by *delta*, starting from the entry whose filepointer equals to **endpt**.
3. Modify the **table of components**. Insert a component entry which corresponds to the new component. The fields of the **table of components** are updated as shown in Figure 4-18.
4. Cut out a region from the container. The region starts from the real filepointer which corresponds to the **startpt**. It ends at the real filepointer which corresponds to the **endpt**.

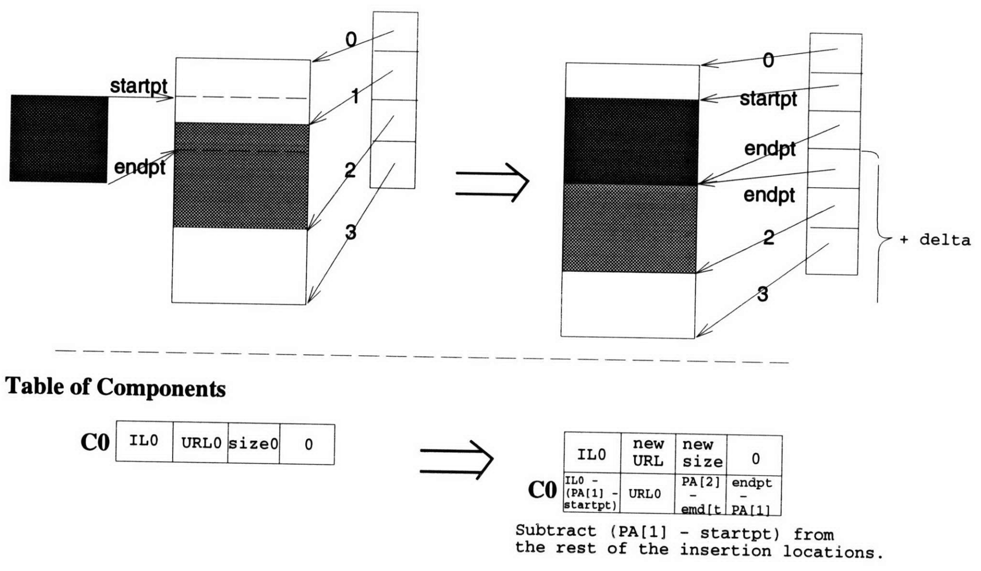


Figure 4-17: Case 3.

5. For all subsequent component entries, shift their insertion filepointers by the size of the cut-out region in Step 5.

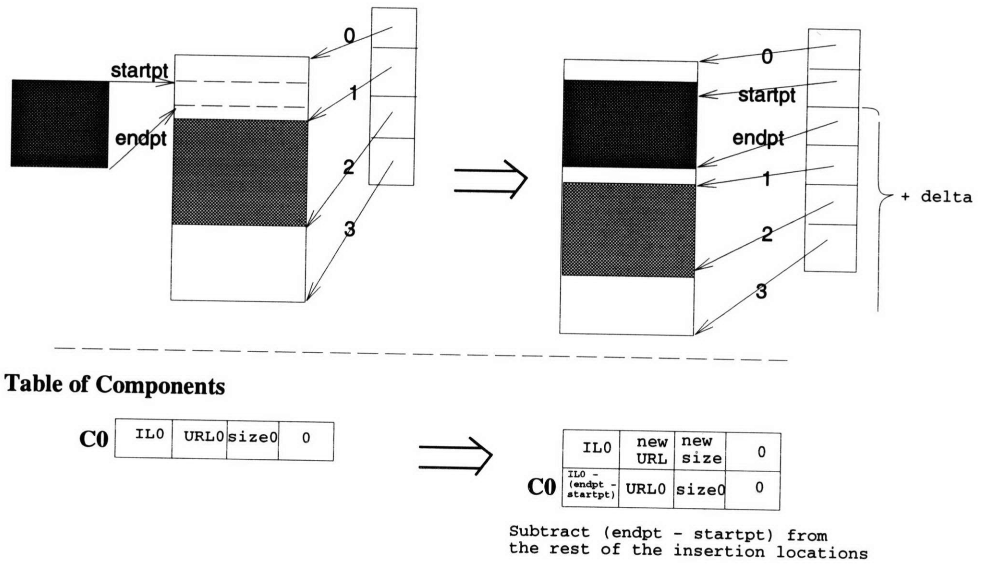


Figure 4-18: Case 4.

4.6 Conclusion and Unresolved Issues

This chapter discusses the implementation of the book paradigm system. The chapter presents the main algorithms and suggests alternate algorithms. The built system matches the specifications in chapter 3. A user may request any page or chapter to be displayed. He may edit a chapter and replace the old chapter with the new edition. All these functions work properly with good performance.

In building this system for purposes of demonstration only, some issues have not been addressed. For example, in the current implementation, the size of a component is retrieved from a lookup table. However, the size of a component may change dynamically. Moreover, a component may not notify the change to any of its containers because a component need not be aware of its containers. This issue can be dealt with by computing the size of components recursively. Thus, if a component does not contain any component, return its file size; otherwise, return the sum of its file size and the sizes of all its components. This algorithm works well for scheme 2. For scheme 1, this algorithm needs to be extended because a component may not be a whole file.

Another possible extension to the system is a distributed lock management. Lock management is needed to provide consistent shared data for users. Though lock management is well established in a transaction processing environment [5], it may be different inside a model of composite objects. One major issue is the granularity of locking [6]. If the entire document is the item to lock, much flexibility is restricted. If a chapter or a page is the item to lock, the item may change even if it is not modified. For example, a page may change when someone modifies another page. Thus the issue of lock management may be very complex.

Chapter 5

Conclusions

We conclude this paper with a brief summary of the ideas that have been presented and a review of further research on the composite model.

We started by discussing the Information Mesh Project, its goals, and its role object model, to give an understanding of the environment for which we needed a composite model. The composite model extends the current role object model. It provides objects with the capability to be composed of other objects. This new capability should be added without changing the current role object model.

We approached the problem by studying a specific example. From the example, we learned the fundamental issues in extending the role object model. The example we studied is a book object. Its role has the parts *chapter* and *page.* These parts expose the substructures in a book. Intuitively, a book object should be divided into components by pages or by chapters; in general, an object would be divided into components based on its exposed substructures. However, the original design of the role object model allows part-instances to overlap, while the straight-forward composite design may restrict part-instances from overlapping. For example, a chapter may begin in the middle of a page, and vice versa. From this book example, we learned the basic conflict between a composite model and the existing role object model.

A well-design composite model needs to resolve this apparent conflict between composition and the flexibility of parts in roles. Our design resolves the conflict by removing the composition mechanism from the underlying representation of a book role. In the composite model. the underlying representation of a book role remains the same as the non-composite version. It reads a book's content from a file, and provides a book interface. Users can read and write the content by pages or chapters. At the same time, a composite abstraction wraps around a collection of files. These files are combined into a single, virtual file. The underlying representation of a book is able to read from the virtual file as if it is an ordinary file.

We then generalize this design of a book to consider other objects in the Information Mesh. In the general model, composition mechanisms are hidden in the underlying representations of a number of roles. If a role supports composition, it will have two underlying representations, a straight-forward representation and a composite representation. Both representations support the methods on the role. Thus the composition mechanism is not exposed to other objects, who see the objects by the role interface. Another concern is the ability to combine objects which play different roles. The inter-roles composition is possible through the inheritance relationships among the roles. A common ancestor presents a common set of attributes in two distinct roles. Therefore, to combine objects playing different roles, each object is cast into the closest common ancestor which supports composite. Then these objects are combined into a composite object which plays the common ancestor role.

Using the above model, we developed a prototype of a book. The prototype is built using Java and CORBA, two recently developed technologies in distributed computing. The two technologies provide the mechanisms for objects to communicate and to be transported across the network. In the prototype, a client-side applet can be transported to any machine in the network. The applet talks to a book server to access a remote book object. It then calls the methods available in a book IDL. The book IDL is analogous to a book role in the Information Mesh. It presents the interface of a book abstraction. Underneath the book object is a virtual file, composed of real disk files. A virtual file is defined recursively; its components are virtual files. The recursion ends in real disk files. Building the book prototype shows that the previously stated design is indeed effective in developing composite objects.

5.1 Further Research

In this paper, we build a book prototype which allows composition of objects. Though the book prototype functions properly, the composite model may not be effective for some other roles. Certain conflicts may appear in some roles but not in the book role. Thus it is important to test the model against other roles, especially those which will support composite in the Information Mesh.

This paper also proposes a general composite model which extends the Information Mesh Object System. In the extension, a role may have multiple underlying representations, one of which supports composition. As well, inter-role composition is accomplished by combining objects based on their closest common ancestor. While the basic design has been described, an actual system still needs to be built so that the model can be studied in greater details.

Another unresolved issue is distributed lock management. The current design of a composite model is incomplete without a distributed locking mechanism. If part-instances are being read and modified concurrently, then inconsistency may arise. Therefore we need to design a lock management which complies with the composite model. The first step towards building a lock management scheme is to clearly specify what the well-defined behavior is. There are a number of possibilities. One possibility is to restrict modification whenever someone is accessing the book. Another is to restrict modification whenever someone is reading a page. A third possibility does not restrict modification at all. A distinct object copy is created for every user. Each user is notified whenever a new edition comes into existence. There are many other possible specifications. After the well-defined behavior has been specified, the lock management can be designed and built.

5.2 Conclusion

This paper suggests a general model for including a composite capability in the Information Mesh. A book prototype has been built according to this model. While the construction of the prototype is essentially complete, it does not cover every idea in the general composite model. A full-scale implementation is needed to demonstrate the ideas in the general composite model. Further research is also needed in the area of distributed lock management.

Bibliography

[1] David D. Clark, Karen R. Sollins, John T. Wroclawski, and Michael L. Dertouzos. Critical technology for universal information access. Research proposal submitted to ARPA, 1994.

[2] Karen R. Sollins and Jeffrey R. Van Dyke. Linking in a Global Information Architecture. In *Fourth International World Wide Web Conference Proceedings* O'Reilly and Assoc., Dec 1995.

[3] Jeffrey R. Van Dyke. Link Architecture for a Global Information Infrastructure. MIT/LCS/TR-659, June 1995.

[4] Paradigms for universality: Networking in the information Age. Abridged Version of Proposal. Submitted in Support of Work by the Advanced Network Architecture Group, 1991.

[5] Andrew B. Hastings Distributed Lock Management in a Transaction Processing Environment CMU-CS-89-152, May 27, 1989.

[6] Andrew S. Tanenbaum Modern Operating systems. Prentice Hall, 1992.

[7] David Ragett. Hypertext markup language Specification Version 3.0. InternetDraft, MIT/W3C, 1995. See http://www.w3.org/pub/WWW/MarkUp/html3/html3.txt

[8] James Gosling and Henry McGilton. The Java(tm) Language Environment: A White Paper. Sun Microsystems, 1995.

[9] Tim Ritchey Programming with Java. New Riders Publishing, 1995.

[10] Frank Halasz. Reflections on notecards: Seven issues for the next generation of hypermedia systems. *Communications of the ACM*, 31(7), 1988.

[11] Frank Halasz and Mayer Schwartz. The dexter hypertext reference. *Communications of the ACM*, 37(2), 1994.

[12] Object Management Group(OMG). The Common Object Request Broker: Architecture and Specification, Revision 2.0 Object Management Group, July 1995.

[13] Robert Orfali, Dan Harkey, and Jeri Edwards. The Essential Distributed Objects Survival Guide. John Wiley and Sons, INC., 1996

[14] Sun Microsystems Java IDL alpha 2.0 Sun Microsystems, June, 1996. See http://splash.javasoft.com/JavaIDL/pages/index.html

[15] Sun Microsystems Java Remote Method Invocation alpha 2.0 Sun Microsystems, June, 1996. See http://chatsubo.javasoft.com/current/rmi/index.html

[16] Hirano Satoshi HORB 1.2.1 Electrotechnical Laboratory, 1996. See http://ring.etl.go.jp/openlab/horb

[17] Sun Microsystems Java Object Serialization alpha 2.0 Sun Microsystems, June, 1996. See http://chatsubo.javasoft.com/current/serial/index.html