RICE UNIVERSITY

Memento: A Collaborative, Semantic-Based Infrastructure for Building Assistant Applications

by

Donald G. Baker

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Doctor of Philosophy

Approved, Thesis Committee:

G. Anthony Gorry, Professor
Computer Science

Moshe Y. Vardi, Noah Harding Professor
Computer Science

Don H. Johnson, Professor
Electrical and Computer Engineering

Houston, Texas

May, 1997

Memento: A Collaborative, Semantic-Based Infrastructure for Building Assistant Applications

Donald G. Baker

Abstract

Memento is a software infrastructure to support the construction and evolution of *assistant applications* (or *assistants*) that act as adjuncts to the human mind. Each *assistant* embodies an effective understanding of some information domain or problem domain. The assistant employs this understanding to aid a user or user community in the manipulation, transmission, and storage of meaningful information artifacts from the assistant's domain, called *models*. Additionally, models flow via the assistant's user interface at the semantic level. Memento provides mechanisms which address all of these common assistant responsibilities. When combined, the mechanisms result in *real–time artifact–based collaboration* over an assistant's models. Memento preserves the meaning of an assistant's models throughout these processes by enforcement of the domain's membership rules, called *semantic constraints*. Because Memento abstracts an assistant's models and semantic constraints, it is therefore said to be *domain–independent*.

Memento is composed of two layers. The bottom layer, described in Chapter 2, supports real–time artifact–based collaboration over collections of name–value attribute pairs, called *MObjects*. A dynamic, hierarchical type system over MObjects characterizes what attributes are required to appear on an MObject. Memento facilitates MObject collaboration through mechanisms for coherent transactions over MObjects, notification of dynamic MObject changes, and shared, persistent MObject storage.

Chapter 3 describes Memento's second, higher layer which permits the expression of an assistant's models as *conceptual graphs*, a knowledge representation scheme based in cognitive psychology. Conceptual graphs are, in turn, represented as collections of MObjects. Model meaning is preserved through enforcement of conceptual graph schemata that characterize a domain's semantic constraints.

A developer can easily create a domain–specific customization of Memento, called a *domain representation*. The domain representation becomes a portable, collaborative applications programmer's interface (API) for the assistant's models and domain. The domain representation is then combined with a suitable user interface to make a complete assistant. This process and several example assistants are described in Chapter 4.

The dissertation concludes in Chapter 5 with a survey of related research in the areas of artifact–based collaboration, human–computer interaction, software engineering, and knowledge representation.

Acknowledgments

The Memento infrastructure and this dissertation would not be possible were it not for the help of a great many people. Memento was born and nurtured in an environment that allowed it to successfully mature. Ross Dargahi and Jerry Fowler were key in defining the initial goals and early design of Memento. Ross gets credit for coining the name "Memento," which has no special meaning. Gwyn Guidy, Tom Lytle, and Vram Kouramajian contributed ideas and, more importantly, enthusiasm throughout Memento's growth. Tony Gorry's support of Memento through the years was instrumental in its success. Thanks, also, to all the members of Rice's Information Technology Development group and Center for Technology in Teaching and Learning, for providing a test–bed for Memento.

Portions of the Memento code were written by others. Gwyn Guidy contributed greatly to the Electronic Studio assistant and fully implemented the Elmo assistant. Jerry Fowler wrote the INQUERY Document Indexing Agent. Tom Lytle ported Memento to the Macintosh environment and explored how Memento might be used with OpenDoc. Kevin Cureton wrote the Electronic Studio WWW Gateway agent. James Wong wrote most of Memento's image conversion utilities. Ross Dargahi contributed classes for access control. The WWW Repository was based on work done by David Hyatt.

In their help in the preparation of this dissertation, I would like to thank my committee members, Tony Gorry, Moshe Vardi, and Don Johnson, for their insightful and constructive criticism. Janice Bordeaux helped me tremendously with the organization and proper emphasis of the material. Thanks to Shisha van Horn for editing the dissertation.

I would like to thank my mentors along the way, Scott Warren and Janice Bordeaux. Scott Warren introduced me to the greater context of computing—one that involves the users of the computers. Scott also helped me appreciate the tremendous impact that visionaries can make on the world. Janice Bordeaux taught me about myself and gave me a glimpse of my own potential.

I have had a number of guardian angels help me in ways too numerous to mention: Tony Gorry, Ken Kennedy, Joanne and Paul Wellman, and Iva Jean Jorgensen. Thanks, too, to my father, Bob Baker, and my brother and his wife, Robby and Holly, for their undying support. Special thanks to Jerry Fowler, who gave me a great deal of (often unwitting) encouragement to turn Memento into a thesis topic and to return to graduate school after some disappointing setbacks. Finally, thanks to my biking buddies for helping me stay sane.

Part of this work was funded by the Advanced Research Projects Agency (ARPA).

This dissertation is dedicated to Jeff Wellman, who believed.

Contents

Illustrations

Tables

Chapter 1

Introduction

The computer has revolutionized the way people deal with information. The revolution is far from over, however. Every eighteen months computers have doubled in speed and storage capacity owing to advances in miniaturization, mass production, and materials science. This trend is likely to continue for some time [88]. Powerful, inexpensive computers are becoming ubiquitous, yet system builders have only begun to explore the computer's potential to enrich peoples' lives. Many of the challenges that lie ahead are in the arena of computer software which seeks to harness the computer's potential for specific ends. One such worthwhile end is augmenting human intelligence.

Using computers to augment human intelligence is not a new idea. As early as 1945, Vannevar Bush proposed a system called *Memex* which would allow the user to quickly access and annotate information which was relevant to him [16]. A user of Memex could navigate through cross–references and save his exploratory trails for later recall. Bush did not speak of computers when he proposed Memex, however he correctly identified knowledge accessibility and the interlinking of related knowledge as central themes in the augmentation of human intelligence. Those themes have only grown in importance. In the late 1960s, Doug Englebart's group at Stanford University built a pioneering system, called *Augment*, with the specific purpose of augmenting the human intellect. Englebart's group realized the importance of the user interface to their task at hand. The group invented many new technologies such as showing video on computer screens, hypertext authoring, and several user interface devices, including the mouse [36].

This dissertation does not concern itself with a single system to augment the human intellect. Instead, a broad class of such systems, called *assistants*, is considered. Assistants and their users form an intellectual synergism. A computer can perform lightning fast calculations, manipulate and process information, transmit it to other computers via networks, and store it faithfully for long periods. Humans are intelligent, creative, and passionate. Each lacks the strengths of the other. Together, they can solve problems which neither one alone could solve. This is an old idea that probably originated in science fiction literature.

The primary goal of this dissertation is to improve the technology related to the construction and maintenance of the entire class of assistants. This chapter first expands on the nature of assistants and identifies two key areas of concern. These areas could be called *meaning maintenance* and the *semantic level of the user interface*. During the discussion of these two areas, attention will be given to both an assistant's requirements of the area and the area's state of the art. While this dissertation introduces the concept of an assistant application, many such applications are already in existence. A taxonomy of assistants is given in Section 1.1.1. The chapter concludes with this dissertation's central thesis claim that the general needs of assistants can be met by a single infrastructure called *Memento*. The Memento infrastructure improves the state of the art of construction and maintenance of assistants.

1.1 Assistants are Applications that Deal in Meaning

An assistant is a computer *application*, or complex software system, that aids its users in solving a problem or representing information. Assistants have some level of "understanding" of the problem or information being represented. By contrast, an on–line calculator is not an example of an assistant. While it performs mathematical functions, the calculator requires the user to translate his problem into a mathematical model before it becomes useful. The ideal assistant application would be like the computer on *Star Trek*: Simply ask a question and it supplies an answer. In reality, however, there is no single assistant that can address all problems. Instead, users must rely on a multitude of assistants—each with its own area of expertise. Section 1.1.1 enumerates several assistant categories.

An assistant and its users form an intellectual synergism. An essential property of that synergism is that the meaning of information is preserved throughout. For example, a user may start with a meaningful model of some problem in his mind. Through one or more interactions with the computer, the user effectively transforms that model into an internal representation maintained by the assistant. Later, the assistant may perform some manipulation to the model. Even if the assistant completely transforms the model, the result must still carry meaning. Finally, the results of the manipulation must be meaningfully communicated to a (possibly different) user. If, at any time in this process, meaning is not preserved, the intellectual synergism is lost.

There are two critical areas in an assistant where meaning must be preserved. First, *meaning maintenance* concerns the preservation of meaning once the information is in an assistant's internal representation. Meaning maintenance includes the persistent storage of information, meaningful transformations on the information, and, in the case of collaborative assistants, preservation of meaning during information transmission between assistants. Section 1.2 discusses the issues surrounding meaning maintenance. The second area is the *semantic level of the user interface* which acts as the conduit of meaning between an assistant and its users. Section 1.3 describes the important aspects of an assistant's user interface in the communication of meaning.

The remainder of Section 1.1 is devoted to describing the current state of the art of assistants. Section 1.1.1 develops a taxonomy of assistants and metrics for their categorization. Section 1.1.2 describes current techniques in the construction and maintenance of assistants. The goal of this work is to improve the state of the art of the construction and maintenance of assistants by solving the central problems common to all assistants in a single infrastructure. The Memento infrastructure is the primary contribution of this research.

1.1.1 Assistant Applications are a Diverse Class

Assistant applications are a diverse class. Without real restriction on the assistant's domain or user interface, few limits exist on the possible types of assistant applications. Many assistants can be built with existing technology. In fact, assistants currently exist in a variety of settings. The major limitation on assistant applications is the expense of their construction and maintenance. This research attempts to mitigate this problem through use of a flexible, reusable infrastructure that aids in the construction and evolution of assistant applications. The infrastructure will allow new assistants to be more easily constructed and maintained. Additionally, it may enable the construction of entirely new types of assistants.

Assistants can be categorized on two scales: *participation* and *collaboration*. The *participation* scale measures the extent to which the application participates in the analysis or manipulation of its information. At one end of the participation scale are assistants that merely hold or store the information for their users. Assistants that

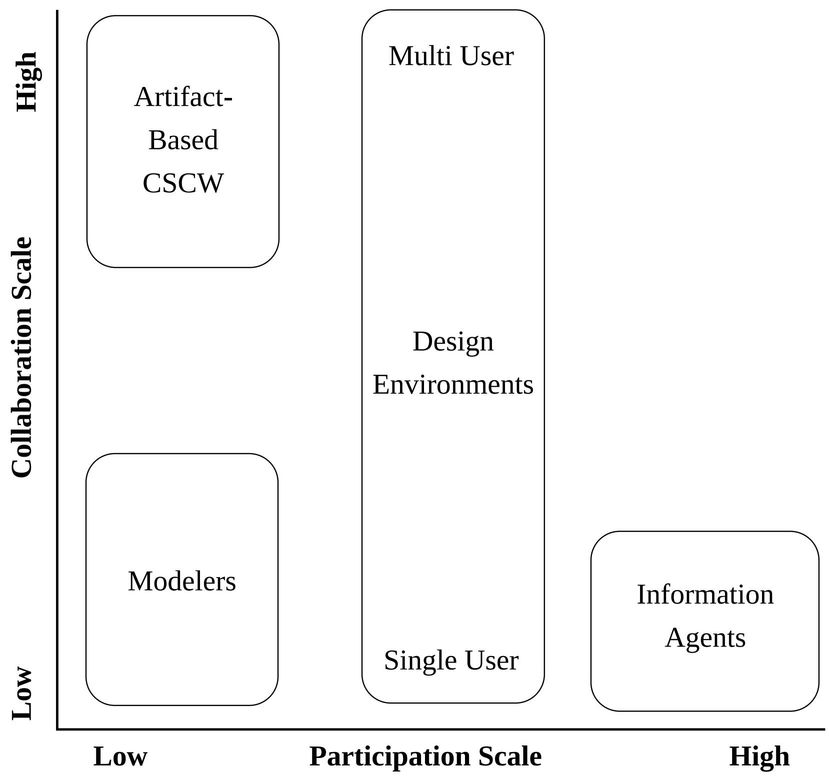

Figure 1.1 Important subclasses of assistants categorized on collaboration and participation scales

analyze, critique, or transform the information are at the other end. The second, *collaboration* scale measures the extent to which an assistant facilitates collaboration among its users. At one end of the collaboration scale are single user assistants. Assistants at the other end of the scale facilitate collaboration among multiple users. These two scales permit a categorization of existing assistants.

Four existing subclasses of assistants include *Modelers*, *Computer Supported Cooperative Work (CSCW) applications*, *Design Environments*, and *Agents*. Each of these subclasses is now discussed in the context of the participation and collaboration scales. Figure 1.1 shows how the assistant subclasses are rated on the two scales.

Modelers simply hold a representation or model on the user's behalf. Modelers are not collaborative, nor do they analyze their users' models. Such applications fall at the low ends of both participation and collaboration scales. Despite their apparent simplicity, modelers can be quite useful. Here, the user may rely on the assistant as an external memory; computers are less forgetful than humans. Perhaps the user creates a model that is so large that he cannot fathom it in its entirety and must focus on one part at a time. Drawing programs and text editors can be considered modelers.

Computer Supported Collaborative Work (CSCW) applications that attempt to address the meaning of the information they manipulate are another important subclass of assistants. Assistants in this class may facilitate group work by some combination of *real–time collaboration*, *workflow*, or *artifact–based collaboration*. In *real–time collaboration*, multiple CSCW assistants linked by a network can play the role of an information switchboard in facilitating the sharing of meaningful models amongst a group of users in potentially distant locations. *Real–time* refers to the fact that all users are made immediately aware of any edits performed by others.[1] The Virtual Notebook System [50] is an assistant that facilitates real–time (and artifact–based) collaboration. CSCW assistants that facilitate *workflow* coordinate the ordering of work done by members of a group. Such assistants may use a mail box or in–basket to notify group members of new work to be done. Lotus Notes [75] and Digital Equipment Corporation's Linkworks [32] are two commercial CSCW applications that facilitate workflow. Finally, *artifact–based collaboration* assistants can facilitate access to meaningful models in a shared information repository. Here, the collaboration need not occur simultaneously or in a strict ordering but can span time as well as distance. Collaboration over meaningful information artifacts has been recognized as an important type of shared work [4, 60]. In artifact–based collaboration, a collection of shared information artifacts can play the role of a group memory that might grow and evolve over a long period. Regardless of the type of collaboration a CSCW assistant facilitates, these assistants are high on the collaboration scale, but because they do not include the computer itself as a participant, they are low on the participation scale.

Design environments are suites of applications where one or more users design something such as a computer network topology, a kitchen layout, or a program [92, 43, 42]. Design environments have domain–specific knowledge of the objects being designed by their users. They can often analyze and critique an emerging design to

---

[1] Note that many real–time CSCW applications are based on the idea of replicating the exact view among all of the users of a session and negotiating for input control. Such applications are often called *WYSIWIS* for What–You–See–Is–What–I–See. Unless the application addresses the meaning of the users' information, it is not considered to be an assistant.

give valuable feedback to the designers. Where a design environment falls on the collaboration scale depends upon whether it is for single or multiple users. Because of their active role in the design process, design environments fall on the higher end of the participation scale.

The last subclass of assistant applications is *information agents*. Information agents typically watch over a changing information space. If a particular condition arises therein, the agent may then notify its user of the condition or summarize information for later retrieval. Information agents are unusual assistants as they need not have a complex user interface. Some information agents simply notify their users of the existence of an exceptional condition. Information agents are generally not collaborative, therefore they rank low on the collaboration scale. Information agents rank high on the participation scale, however, because they often do significant analysis and interpretation of their information.

While assistants are currently a diverse class of applications, future assistants will likely exhibit even more diversity. The limitations of current software technology render many possible assistants too costly to build and maintain. An improvement in technology could enable more assistants in the identified categories; completely new categories might emerge. For example, what type of assistants might be highly participatory and highly collaborative?

1.1.2 Today's Assistants are Hand–crafted

The assistants built with today's technology are directly programmed and extensively customized. An assistant's "knowledge" of the information it manipulates is generally programmed into the assistant for a specific problem domain. By contrast, the field of artificial intelligence may someday offer a method of building assistants which relies on a distilled knowledge representation, perhaps derived from interviews with experts. The futuristic solution would presumably allow the experience of many experts to be combined into one super–assistant. The user interface of such an assistant might rely on a natural language interface. By contrast, today's assistants must have a user interface that is extensively customized for its problem domain.

Since the advent of computers, researchers have given a great deal of attention to augmenting computer intelligence. If the intelligence of a computer could rival man's own, it would make an ideal assistant. Such an assistant would have an implicit understanding of many areas of knowledge, be able to make inferences, and communicate fluidly with its users. Science fiction has long pondered the question of the ideal assistant. Perhaps it would be like the anthropomorphic and omniscient computer on the *Star Trek* television series. An ideal assistant would be a logical consequence of machine intelligence.

The field of *artificial intelligence* is devoted to the problem of making computers more intelligent. Early artificial intelligence research met with surprising successes. During the 1960s and 1970s, systems like the General Problem Solver [38] and MYCIN [101] were solving problems in areas (math and disease diagnosis) thought to be the exclusive domain of human experts. More recent research, however, has forced us to reevaluate our expectations of computer intelligence (and hence, assistants). Even the seemingly simple task of interpreting children's stories proves to be very difficult for the computer. The process requires large amounts of *world knowledge*[2] for interpreting language, making inferences, and understanding context. With an unlimited range of discourse, the requirement for world knowledge becomes an overwhelming obstacle.

Expectations of the ideal assistant's user interface must also be reexamined. Unfortunately, even the transcription of spoken word into text is another manifestation of the world knowledge problem. Natural language dialogue is not an efficient means of communication between man and machine. It is too slow, full of subtlety and ambiguity, and does not make use of the range of human senses. Finally, many areas of knowledge are visual in nature. They would be poorly matched with a natural language interface.

The dream of an omniscient, anthropomorphic assistant must, for now, remain a dream. If the expectations for useful assistants are scaled back, however, qualifications can be derived for assistants that can be built with current technology. First, the universe of knowledge of the assistant must be limited so that the artificial intelligence problem of world knowledge can be avoided. Instead of some generalized intelligent knowledge engine, each assistant's "knowledge" in some domain is directly programmed (as opposed to relying on knowledge rules, such as in expert systems). As such, each assistant becomes something of an idiot savant in its particular area of expertise and cannot be considered truly intelligent. Second, the user interface must be tuned to facilitate efficient communication between the user and assistant

[2] World knowledge is the myriad facts and stereotypic assumptions that humans take for granted, such as "dogs have four legs" or "piggy banks hold coins." Within a program, such knowledge must be explicitly captured and represented.

concerning anything in the range of knowledge of the assistant. Instead of a single one–size–fits–all user interface that allows the user and assistant to communicate about anything, each assistant must have its own customized user interface.

1.2 Domains and Models Capture Meaning

Meaning maintenance is the first of the two areas of concern to all assistant applications. *Meaning maintenance* includes the representation, manipulation, transmission and storage of information. In all cases the operations must be meaning–preserving. This section will explain how a single infrastructure can fill the needs of diverse array of assistants in the area of meaning. First, the purpose of each assistant leads to an information domain. Next, domains can have several different physical representations. Domains can be described by a set of rules called *semantic constraints*. The last two subsections enumerate the requirements and benefits of a single generalized model and domain representation and infrastructure.

1.2.1 A Domain is Derived from an Assistant's Purpose

The purpose to which an assistant is applied is intimately tied to the type of meaning the assistant can handle and manipulate. If an assistant rates high on the participation scale, the assistant's purpose derives from a *problem domain*. A problem domain is the set of items to which a particular assistant's problem–solving or analytical abilities can be applied. If, instead, the assistant is low on the participation scale, the assistant's purpose derives from an *information domain*. An information domain is the set of items that may be represented, stored, or participate in a collaboration, but for which there is no expectation that the assistant will perform significant manipulation. The generic term *domain* encompasses both information domains and problem domains.

An assistant's *domain* is its effective range of meaningful information discourse. It is the range of the assistant's "knowledge." Items in the domain are meaningful information groupings called *models*. A domain can be thought of as a space of possible models. Models originate in the mind of a user, perhaps in the form of mental models [85, 20]. In order to take advantage of an assistant's abilities, a user must be able to communicate a model to an assistant. When this happens, the model becomes the subject of a conversation between the user and assistant. How models are communicated via the assistant's user interface is the subject of Section 1.3.

An assistant can be thought of as an environment for various users and computer agents to interact with *models* from its domain. An important feature of the environment is that the meaning of models is preserved throughout; the assistant must maintain the semantic integrity of its models during all manipulations, including storage, transformation, and transmission to other assistants. Additionally, the assistant's user interface must faithfully communicate its models with its users.

1.2.2 Semantic Constraints Define a Domain

Within the context of a domain, there are constraints on what kinds of models are meaningful. The constraints can be thought of as a set of rules that determine a model's inclusion in the domain. Only those models which satisfy the domain's constraints belong to the domain. Such rules are called *semantic constraints*. Semantic constraints must exist in some form wherever models from a domain are to retain meaning. Semantic constraints must exist in the minds of users, be embodied in the assistant's code, and be shared among the designers of the assistant. Ideally, the rules should specify the same constraints regardless of their form.

Within a user's mind, for example, the semantic constraints exist in a mental representation. A user's representation may be based on a familiar problem domain. Alternatively, a domain may be based on a metaphoric resemblance to a "real world" situation. In such a case, the assistant may employ a user interface metaphor to help its users map the "real world" onto the target domain. The semantic constraints of a domain can be extracted from a single expert or representatives of a user community by means of knowledge elicitation techniques [25].

Assistants are the result of conscious design by software engineers. Designers, therefore, must have a means of formally describing domains and their semantic constraints as part of an assistant's design. By using an Object–Oriented Analysis and Design (OOAD) methodology, for example, the semantic constraints for an assistant's domain become part of the end result of the design process. Often, this result takes the form of diagrams and a formalized notation for capturing and communicating the design such as semantic networks, petri–nets, or state diagrams [7].

Once an assistant is built, it becomes a programmatic embodiment of its domain. Here, the semantic constraints are hard–coded into the assistant application. The hard–coded semantic constraints are used during all of the assistant's key processes. The assistant uses the semantic constraints to enforce the correct construction and manipulation of models via the user interface. Additionally, semantic constraints come into play during the manipulation of models and model transmission to other assistants. Finally, the assistant may use its programmed semantic constraints in an indirect way to educate its user community about its domain. One way to judge the quality of the assistant's user interface is by how well users can acquire the semantic constraints of the assistant's domain through interaction with the assistant.

Domains and their semantic constraints are not static; they usually evolve over time. An assistant's information domain may evolve in some unanticipated way, perhaps due to some change in the reality the assistant is modeling. If an assistant is successful, it may be redeveloped for another platform or a different user community. Eventually, a suite of assistants may be built around a group of information domains related to the original. All of these scenarios require changes to the code of the original assistant.

When an assistant's domain must evolve, the hard–coded semantic constraints must be modified. If the assistant is well designed and documented, modifying programmatic semantic constraints is relatively straightforward. If the assistant has undergone a great deal of change, or if it is distributed, as in the case of collaborative applications, modifying its semantic constraints can be difficult and error–prone. Eventually, the assistant becomes too expensive to maintain and is rendered obsolete. Sadly, if such an assistant is to be re–developed, the same arcane complexity that hindered its maintenance also inhibits reuse of its code in the next version. The entire effort in coding the original assistant can be lost. The issue of assistant evolution is addressed by specific mechanisms described in this dissertation. This current state of the art does not lend itself to the evolution of a domain's semantic constraints.

1.2.3 Requirements of a Generalized Model and Domain Representation and Infrastructure

In order for a variety of assistants to benefit from a single infrastructure, the models and domains of the assistants must have a common representation and share the same mechanism for meaning maintenance. The representation becomes the basis for the infrastructure upon which assistants are to be built. For the representation and infrastructure to be effective and useful, they must meet requirements of:

- domain expressivity,
- self–containment of semantic constraints,

- flexible semantic constraint enforcement, and
- other practical considerations.

Each requirement is explored in turn.

An *expressive model and domain representation* means that a large variety of assistants' domains can be expressed. Expressivity is difficult to quantify, but one way to demonstrate it is to build a diversity of example assistants' domains using the representation. A better way is to give a method for eliciting a domain's semantic constraints and expressing them in the representation. In the former case, it can be argued that a completely new assistant would not be significantly "different" than the examples. In the latter case, determining and describing the boundaries of expressiveness is more likely. After the Memento infrastructure is described, the issue of expressivity is again discussed in Section 3.6.4.

Self–containment implies that the "knowledge" about the domain is in a single package. In order for an assistant to effectively enforce the semantic constraints, the package of constraints must be embodied in the assistant. This embodiment is called a *domain representation*. Ideally, nearly all of the enforcement can be delegated to the domain representation (and infrastructure). When the semantic constraints can be so contained, their maintenance and evolution are greatly facilitated. Additionally, if the semantic constraints are also declarative, the infrastructure can enforce multiple domains simultaneously. Declarative semantic constraints can be stored along with the models from the domain and shared among assistants that collaborate over the domain, thus facilitating distributed meaning maintenance.

Next, the infrastructure must take the semantic constraints and assume the responsibility of enforcing those constraints over models in the assistant's domain. The infrastructure must be flexible in how it enforces those constraints. It must allow the constraints to be violated temporarily during the models' manipulation and re-verified upon completion. The infrastructure should allow all manner of manipulations as long as the semantic constraints can eventually be validated.

Finally, the representation and infrastructure must be appealing to software designers. It must be relatively easy to use and yield higher quality assistants, less expensive assistants, or both. The infrastructure must allow for customization of semantic constraints that cannot be completely captured by the declarative notation. The infrastructure must be portable to a variety of platforms; real–world applications are not often implemented on academic architectures. Additionally, the representation and infrastructure must allow for graceful evolution of the domains they represent. Section 1.3 identifies the assistant's user interface as an important source of evolutionary change in an assistant.

1.2.4 Benefits of a Generalized Model and Domain Infrastructure

Once the requirements for a generalized model and domain infrastructure are met, the benefits are manifold. The benefits include reuse of code, ease of assistant construction, and ease of assistant evolution. Each of these benefits is now discussed in turn.

A generalized model and domain infrastructure can enable reuse at three different levels. First, the infrastructure itself can be reused in many assistant applications. Next, a domain representation constructed from the infrastructure can also be reused in related assistants. The infrastructure's domain representation allows the semantic relationship between two or more related information domains to be formally and explicitly expressed. For example, if two information domains share a common sub-domain, the sub-domain's domain representation can be shared by reference from the original two domains' domain representations. When such an association is made explicit, the sub-domain can have a life of its own and evolve through time. Finally, while making use of the infrastructure, there is a potential to design the user interface for an information sub-domain in a reusable fashion and thus facilitate the evolution of suites of assistants that manipulate related information domains. User interface code associated with the shared sub-domain can potentially be shared among all applications that use that sub-domain. Reuse is greatly facilitated if the components are designed from the outset to be used again [65]. Reuse of high–level user interface components is currently difficult without a clean separation of responsibilities that would be imposed by a generalized model and domain infrastructure.

Because the infrastructure solves the difficult problem of collaboration over an assistant's models in an generalized way, collaborative assistants can be more easily constructed using the infrastructure. Collaborative systems over specific domains are generally regarded as difficult to build from the ground up [35]. A generalized infrastructure can solve the difficult problem of meaning maintenance: the representation, storage, coherent manipulation, and transmission of models from each assistant's domain. The effort required to customize the infrastructure for a representation of an assistant's domain is minute compared to the effort of developing the corresponding domain representation without such an infrastructure. The infrastructure described in this dissertation is parameterized for each new domain; hence it is domain–independent. Section 5.1.2 surveys a variety of CSCW toolkits and relates them to the assistants' concerns.

Assistants built using a generalized model and domain infrastructure are more easily adapted to evolutionary changes in the assistant's domain. Because the assistant's domain representation is mostly declarative, it can be readily modified. Properly constructed assistants can delegate semantic constraint enforcement to the domain representation, hence little user interface code must be changed when the domain evolves. Assistants built using a generalized infrastructure may have a longer life as they are more insulated from the fundamental changes associated with domain evolution. Longer lived assistants are ultimately more economical.

1.3 Meaning Flows through the User Interface

The second of the two major areas of concern for all assistant applications is that of the user interface. An assistant's user interface must act as a conduit of meaning between the user and the application. In order to understand how this is done, the reader must first understand what is meant by the semantic level of the user interface. Subsections 1.3.1 and 1.3.2 discuss this concept and its importance.

This section will later explain how the same generalized model and domain infrastructure can improve the state of the art in constructing user interfaces for assistants. In this situation, however, the benefits are not as obvious or as great. User interfaces remain a challenging area of application construction for which no simple solutions exist. Subsection 1.3.3 discusses the current state of the art in user interface construction. The final subsection discusses the specific benefits that the infrastructure would provide. Many of these benefits are not immediate, but accrue over time as the assistant evolves.

1.3.1 The User Interface has Semiotic Levels

The main theme in Computer–Human Interaction (CHI) literature is that the user interface is a communication medium between the program and user. One can speak of the language—primarily visual and tactile—between the executing program and the user sitting in front of the machine. Like any language, the user interface can be broken down into its semiotic levels: lexical, syntactic, and semantic [78]. The lexical level of the user interface deals with things like font selection and menu layout when the program is communicating to the user or, when the human is communicating to the program, events like key strokes and mouse clicks. The syntactic level is more concerned with how the user invokes operations. For example, the user may make a selection, which becomes the subject of an operation (a noun) and then chooses an item from a menu indicating the operation to be performed (a verb). The most interesting level of communication, however, is the semantic level where the user and program are potentially engaged in a meaningful exchange.

Within the semantic level of the user interface the computer communicates what it "knows" and what is possible; the user communicates what is to be done. The semantic level of the user interface may not have a tangible appearance or behavior in the user interface. Instead, it is the gestalt of the lexical and syntactic layers that is better described in terms of the application's facility in a particular information domain. The quality of a user interface is largely judged by how well that semantic exchange is made or, conversely, by how little the lower levels of communication interfere with that exchange.

Great advances have been made in the study of the lexical and syntactic user interface levels. It is relatively straightforward to study usability on menu orderings, icon shapes, window layout and the like. From these studies, good user interface guidelines and standards have emerged [100]. When a user interface standard is adopted by a vendor, it can be embodied in a user interface tool, such as an application framework or user interface toolkit. Apple Computer's MacApp is one example of an application framework that embodies the Macintosh user interface standard [95]. Such tools make it much easier to develop applications that adhere to the embodied standard.

The semantic level of the user interface is far more difficult to study. Only a few research groups have addressed it. Researchers at Xerox have undertaken a program to explore how user interface metaphors lend themselves to various general information domains [17, 93]. Their goal is to develop a rich vocabulary of reusable metaphors. A group at George Washington University explored separating the semantic level of the user interface from the more concrete levels so that they may be parameterized for different user communities [39, 48]. In their research, the semantic layer of the user interface is not varied. Other groups have explored the notion of reusable user interface widgets that convey semantic information via familiar visual information organizations [58, 84, 111, 61]. They have investigated tables and outlines as potential bridges between the syntactic and semantic levels of communication in the user interface. Such research dovetails nicely with the notion of a domain–based infrastructure such as that discussed in this dissertation. Often there is a close connection between the visible user interface metaphor of an application and its information domain [19].

Research into the semantic level of the user interface has lagged behind that of the other levels. As a result, there are no tools that aid in the development of the semantic level of the user interface. The infrastructure and associated methodology developed in this dissertation could be considered the first such tool.

1.3.2 The Semantic Level is Crucial

While the semantic level of an application's user interface is the most elusive level, it is also the most important. It is the semantic level of the user interface that by–in–large determines the usefulness and longevity of an application. Lexical and syntactic problems in a user interface are usually easily correctable, but an ill–conceived semantic level is often a fatal flaw. Unfortunately, it is the semantic level of the user interface that is the most difficult to design, build, and modify. It is where the real art lies in the construction of user interfaces.

When the scope of programs is limited to assistants, much more can be said of the semantic level of the user interface. The range of discourse is precisely the set of models from the assistant's domain. The user interface must convey the domain's semantic constraints to the user and the user must be able to manipulate models in meaningful ways. Ideally, the user must have the feeling that he is interacting directly with the assistant's models. Fischer and Lemke have used the term *human–problem domain communication* to describe this idea [43, 41]. Their ideas apply just as well to assistants' information domains. They make the point that there is currently too little focus on this deeper level of communication in the construction of user interfaces. This dissertation adopts this idea as a central theme.

1.3.3 User Interface Construction is Iterative and Expensive

Assistants belong to a class of computer systems, called *applications*, which are programs that embody a high degree of complexity. Like most applications, assistants generally have a significant user interface component. For example, it has been estimated that one–third to one–half of an expert system application's code size is devoted to its user interface [9]. There is no reason to expect that this estimate would not generalize to the broader class of assistants. Unfortunately, user interfaces take a disproportionate share of development time and effort. Unlike most software projects, a user interface cannot be specified in advance. This means that traditional software development methods do not help. There are no fool–proof procedures a developer can follow *in isolation* that will result in a good user interface.

The only proven way of developing high quality user interfaces is through iterative development and user testing with each cycle feeding into the next [52]. User testing involves placing people who have similar aptitudes and training as the target user group in front of the trial application; capturing their responses, errors, comments, and frustrations. Later, these results are analyzed with an eye toward determining the difficulties' root causes. A new user interface is then generated which attempts to remedy the problems and user testing begins anew. Ideally, iterative development continues until some pre-determined criteria of user speed, error rates, or proficiency is reached. In practice, economic constraints usually prevail.

The user interface often evolves in unforeseen ways during iterative development. It can place a stress on the quality of an application's code. Developers have tried to mitigate the stress by prototyping facade user interfaces for evaluation. The light–weight prototypes can be generated quickly at each step in iterative development with only small compromises in realism. Ideally, when the prototype is accepted, the final application is re-coded based on the prototype. Such a development program is expensive, however, and it is often the case that the prototype's "throw away" code ends up in the final application.

1.3.4 The State of the Art Can Be Improved

The generalized infrastructure for model and domain representation can improve the state of the art of the construction and evolution of assistants' user interfaces. The benefits of the infrastructure are largely derived from the embodiment of the assistant's semantic constraints in a single *domain representation*. The domain representation becomes the primary component of the semantic level of the user interface and the focal point of the user interface as a whole. The domain representation aids in both the initial construction and evolution of an assistant's user interface.

When the infrastructure is used, user interface construction is aided by the fact that the semantic level is much easier to build and this helps the developer separate concerns. The domain representation is based on a reusable infrastructure that is specialized in a declarative fashion. The enforcement of the semantic constraints can be relegated to the domain representation and need not be duplicated elsewhere in the user interface code. With the domain representation complete, developers can concentrate on constructing the concrete levels of the user interface: the syntactic and lexical levels. Once the user interface is constructed, it can be reused in every assistant that shares its associated domain.

The evolution of the user interface is facilitated by the use of the infrastructure. Although the semantic and higher levels of the user interface are intimately tied, they are decoupled as much as possible in an assistant built with the infrastructure. The higher levels can change radically for a given domain without any need to modify the semantic layer. In fact, multiple user interfaces can be built on the same domain representation for different user communities. If the semantic constraints of the domain change, the domain representation can be quickly adapted. With the higher levels of the user interface relying on the domain representation, for constraint checking, the impact to the higher levels of the user interface is minimized.

1.4 Thesis Statement

The thesis of this dissertation is that a domain–independent infrastructure for representing, storing, transmitting, and modifying models from diverse information domains while maintaining the their semantic integrity would improve the state of the art of building and maintaining assistants in the following ways:

- The infrastructure would abstract the problem of representing, transmitting, storing, and manipulating models from various information domains. The infrastructure would be built once and reused for a large variety of assistants.

- The infrastructure would abstract the problem of meaning maintenance, the enforcement of semantic constraints from diverse information domains. The abstracted *domain representation* would be easier to create and modify than a hard–coded one.

- The infrastructure would simplify the creation and ease the evolution of assistants' user interfaces. An assistant's domain representation is the foundation of the semantic layer of its user interface. The remainder of the user interface is decoupled as much as possible from the domain representation.

- The infrastructure would facilitate domain–independent collaboration. Domains are represented by declarative semantic constraints.

- The infrastructure *might* enable cross–domain interoperability and collaboration through shared information sub-domains. The semantic constraints from multiple, related domains could be simultaneously enforced.

- The infrastructure would aid in the reuse of domain representations and *might* aid in the user interface code associated with shared information sub-domains. A deeper level of reuse could possibly be enabled.

- The infrastructure *may* facilitate the construction of entirely new types of assistants.

These thesis claims will be validated using the *Memento* infrastructure. The Memento framework and associated methodology addresses all of the above mentioned improvements in the state of the art. Memento uses a flexible, domain–independent knowledge representation scheme for abstracting models from diverse information domains. Memento uses the knowledge representation scheme to represent, transmit and store models. Memento allows the expression of an assistant's semantic constraints by way of a declarative notation over the knowledge representation scheme. Memento uses the declared semantic constraints to enforce meaning–preserving manipulation of an assistant's models in a domain–independent way. As a portable framework, Memento allows for the quick generation of a customized, portable information domain representation that can later be easily changed when the semantic constraints of the domain require modification. When combined with an appropriate application framework, the portable domain representation forms the basis of an assistant's user interface. The separation of the semantic constraints into the domain representation allows the constraints to evolve as independently as possible of any user interface based on the domain representation. Memento attempts to mitigate the amount of re-coding required when the user interface or information domain needs to change by encouraging a style of programming that separates concerns, insulates components from change, and encourages reuse. Memento facilitates collaboration over models in a way that is independent of their domains. As a result, Memento facilitates domain–independent collaboration and may facilitate cross–domain collaboration. Finally, when sub-domains are shared across assistants, Memento may permit the user interface code associated with that sub-domain to be shared and reused when assistants are built that are based on the sub-domain.

In order to justify the thesis claims, a small number of experimental, Memento–based assistants are developed as a proof of concept. These assistants demonstrate:

- how the infrastructure facilitates human–problem domain communication in the user interface of each assistant,
- how the infrastructure supports collaboration over models in a domain–independent way,
- how Memento facilitates the evolution of assistants and their associated information spaces, and
- how Memento can support information agents.

1.5 Overview of the Dissertation

The remainder of this dissertation is organized as follows. Chapters 2 and 3 describe the two levels of the Memento infrastructure in a bottom–up fashion. The lower level, described in Chapter 2, deals with the representation, storage, transmission, and collaboration over generalized attributed objects. Chapter 3 describes the second level of Memento that addresses model representation and semantic constraint enforcement. Chapter 4 gives a methodology for constructing an assistant's domain representation. The chapter also describes several experimental assistants constructed with Memento and the lessons learned from them. The final chapter summarizes the contributions of this dissertation and compares these contributions with previous work done by others.

Chapter 2

Memento's Layer I: Generalized Artifact–Based Collaboration

Conceptually, the Memento infrastructure is divided into two layers that address orthogonal issues. Memento's lower layer supports artifact–based collaboration over *Memento objects* or *MObjects*. An *MObject* is simply a collection of name–value (attribute) pairs. Memento's first layer is the subject of this chapter. The second layer of Memento, the subject of Chapter 3, supports the semantic needs of assistants. The chapter describes how an assistant's models and domains are represented as specialized graphs built from MObjects.

2.0.1 Memento Must Support Artifact–Based Collaboration over Models

Each assistant is an application that manipulates models from its domain. Every model has a meaning and a purpose for its existence. In the physical world, man–made objects with meaning and purpose are called *artifacts*. Models are intangible, however. They may only exist within an assistant and in the minds of its users. Models are artifacts, nevertheless. To distinguish them from physical artifacts, they are termed *information artifacts*. (Note that not all systems that deal with information artifacts are assistants. In order to be considered an assistant, such a system would need to have an effective understanding of its information artifacts.)

As discussed in Chapter 1, many assistants facilitate collaboration over their models. If Memento is to serve as an infrastructure for assistants, it must support collaboration over models from a wide variety of domains. Fortunately, the infrastructure needs of non–collaborative assistants, such as modelers and design environments, are a subset of the needs of collaborative assistants. These needs include persistent model storage, and communication of models between assistants and through an assistant's user interface.

Memento is a contribution to the ongoing study of *artifact–based collaboration*, which can be defined as the coordinated effort of a group of people to produce a result or artifact. The group authoring of a document is an example of artifact–based collaboration. Video teleconferencing, on the other hand, is a form of collaboration that is not artifact–based. While a video record of the collaboration may be kept, the production of the video record is not the purpose of the collaboration.

2.0.2 The Advantages of a General Artifact Representation

At issue is the nature of the artifacts over which the collaboration is performed. Previous systems supporting artifact–based collaboration do so for a wide range of specific information artifact types including text documents, webs of hypertext documents, outlines, calendars, network designs, white board drawings, and issue–based reasoning structures. In order to provide support for a wide variety of assistants, Memento's artifact representation must be very general and adaptable. The choice of representation determines, to a large extent, the set of assistant applications that can be built with Memento.

The intent of Memento is to support collaboration over an arbitrary assistant's models, however models are not the fundamental unit of collaboration in Memento. Instead, a simpler fundamental unit of collaboration was chosen: a collection of name–value attributes, called an *MObject*. Each assistants' models are represented in terms of MObjects. This design decision has two very important consequences. First, an infrastructure that can fully support collaboration over MObjects effectively solves the artifact–based collaboration problem "once and for all" in an abstract way. Using just Memento's first layer, a developer can effectively build any existing artifact–based collaborative system (whether or not such a system is considered an assistant that deals with the meaning of the artifact) by simply representing the system's artifacts in terms of MObjects. The issues of artifact–based collaboration are generalized and solved by Memento's first layer. A discussion of artifact–based collaborative systems and how Memento contributes to this area is given in Chapter 5.

The second implication of the decision to support collaboration over MObjects is that more than just an assistant's models will be built from them—an assistant's meta–information will be constructed from MObjects, as well. An assistant's meta–information consists of a domain representation that is built upon a type system over MObjects.[3] Recall from Section 1.2 that a domain representation captures the

[3] Because Memento's type system over MObjects is expressed in terms of MObjects, it is self–describing and self–referential.

semantic constraints over an assistant's models. Within Memento, a domain is represented by a set of *metamodels*, built from MObjects, that effectively give meaning to the assistant's models. Metamodels are stored and communicated along with an assistant's models in order to preserve a context of meaning. For example, if two assistants cannot reach a consensus over the type system and metamodels describing the models to be shared, collaboration would be meaningless and could not proceed. Because models, the type system over MObjects, and metamodels all share the same fundamental representation, few additional mechanisms are required for manipulation of an assistant's meta–information. Note that the evolution of assistant's domains is greatly facilitated by the fact that Memento represents the bulk of an assistant's meta–information as data.

Section 2.1 describes the representation of MObjects and the type system over MObjects. Section 2.4 describes how an assistant's semantic constraints give rise to a type system over MObjects, how an assistant's classes relate to specific types, and how separate type systems can be unified when models are to be shared across MType systems. The details of the remainder of the domain representation (i.e. metamodels) and how it is used by Memento to address the semantic needs of assistants is left for Chapter 3.

2.0.3 Required Mechanisms for Artifact–Based Collaboration

Once the nature of the artifact is abstracted, all systems to support artifact–based collaboration deal with largely the same set of issues. These issues are how the artifacts are coherently modified, how changes to the artifacts give rise to changes in the user interface, and how the artifacts are persistently stored and communicated. The remainder of this chapter discusses the Memento mechanisms that perform these tasks. Specifically, Section 2.2 describes Memento's transaction mechanism over MObjects which allows for coherent, atomic modification of sets of MObjects. Section 2.3 describes the mechanism whereby user interface objects can receive notification of changes to the MObjects they are displaying, and thus update the display accordingly. Section 2.5 describes how MObjects are persistently stored in an *MObject repository*. Communication between assistants is done via a special *network repository* that treats one assistant's MObjects as a repository for another. Taken together, these mechanism comprise a general framework for collaboration over MObjects. An example of how the collaboration support mechanisms work together during a run of a typical collaborative application is given in Section 2.6. The chapter concludes with Section 2.8 which describes the implications of Memento's first layer.

2.1 The Memento Object (MObject) Data Model

Memento Objects, or *MObjects*, are the primary unit of information sharing and collaboration of Memento's first layer. MObjects are collections of name–value attribute pairs. Within Memento, an MObject exists as an instance of the `MObject` class.[4] A type system, called the *MType* system places constraints on the attributes that an MObject must have. The MType system is built from a separate type system over attribute values, called the *transport* type system. An assistant's type system is represented as a tree of MObjects. It is therefore self–describing and readily modified. This section concludes with an example of a set of consistent MObjects showing the MType system.

2.1.1 MObjects are Collections of Name–Value Pairs

An MObject is a collection of attributes where each attribute is a name–value pair. An attribute's name is unique in an MObject. Each attribute's value is drawn from a set of *transportables*. The type system over transportables, called the *transport type* system, is discussed briefly later in this section. Operations on an MObject include its creation and destruction, creation and destruction of an associated attribute, enumerating the names of its attributes, and accessing and changing the value of an existing attribute. All high–level operations over MObjects are combinations of these few fundamental operations.

Combinations of the basic operations on MObjects occur in the context of a transaction. Like database transactions, a Memento transaction has the features of atomicity and validity checking. *Atomicity* implies that if the transaction succeeds, all operations permanently take effect. Whereas, if the transaction fails, none of the operations take effect. Success of a transaction is determined in part by whether the transaction passes its validity checks at transaction commit–time. For the sake of Memento's first layer, a transaction over a set of MObjects is valid if the Memento type system is not violated. The Memento type system, described below, dictates what attributes must appear on an MObject. Chapter 3 extends the notion of validity

[4] A monospace font is used when discussing concrete components of the Memento infrastructure.

checking during transactions to ensure that models, which are built from MObjects, do not violate the semantic constraints of the assistant's domain.

During the course of a transaction, notification is given to software components that have registered an interest in changes to a particular MObject. Notifications are given out both during each of the trial modifications of a transaction and at transaction commit–time to summarize the modifications that actually happened, or, in the case of a commit failure, the trial modifications that should now be reverted. Change notification is the mechanism by which changes to MObjects are propagated to the user interface. Memento's notification mechanism is the subject of Section 2.3.

2.1.2 MObjects have a Memento Type (MType)

Every MObject has a *Memento type*, or *MType*, that delineates what attributes are required to appear on that MObject. Required attributes are specified abstractly by the attribute name and the transport type to which the attribute value must conform. Specifically, a *requirement* is an attribute name, the transport type of the required value, and an optional default value having that transport type. Of course, the default attribute value is ignored for those MObjects that supply an overriding value. A description of the transport type system over attribute values appears below.

The set of MTypes form a singly–rooted type hierarchy.[5] An MType effectively adds to the set of required attributes of its supertypes. Therefore, an MType need only represent the difference in requirements between it and its supertype. More formally, an *MType* consists of a name, an optional supertype, a set of subtypes, a set of requirements, and a set of references to MObjects that have exactly that MType. The supertype and set of subtypes form a doubly–linked type hierarchy representation of the MType system. MType names must be globally unique. Required attribute names appearing on an MType must be unique along any possible path through that MType along the way to the root MType. This is consistent with the constraint that an MObject not have two attributes of the same name.

The Memento transaction's validity check ensures that each MObject involved in the transaction has the attributes required of their associated MTypes and that the value of each required attribute is a member of the requirement's transportable type. This check must also succeed if the MType of an MObject changes during the course

[5]While there are several reasons why it might be desirable to have a more general partial order over types, the simpler representation was explicitly chosen.

of the transaction. An MObject can have additional attributes that are not required by the MType system. Such attributes can serve as annotations and do not take part in the transaction's validity check.

Attribute Values have a Transport Type

An MObject attribute value is known as a *transportable*. The name derives from the fact that all attribute values can be serialized for transport between address spaces or for storage. The type system over transportables is called *transport types*, or *TTypes*, for short. The set of TTypes is open. While Memento includes a fairly comprehensive set of TTypes, there is always the possibility that a domain will require additional ones. A programmer can add additional transport types as part of his assistant and they will behave as if the were part of the original set.

TTypes are categorized as to whether they are *primitive*, *aggregate*, or *special*. *Primitive* transport types include Boolean, Count, String, and more complex types like Blob (large binary object), RasterImage, and DateTime. *Aggregate* TTypes are arbitrarily complex groupings of simpler TTypes. Aggregates include sets, lists, and records. *Transportable sets* are unordered, non–duplicated collection of transportables of a given TType. *Transportable lists* are ordered collections of possibly duplicated transportables of a given TType. A list TType additionally specifies a cardinality range over its members. Like its counterpart in many programming languages, a *transportable record* is an association between names and values. A record TType is an association between names and the TTypes of the expected values. *Special* TTypes include TAny, which is the supertype of all TTypes, and TUnknown, which acts as a placeholder for TTypes that an assistant cannot represent. Table 2.1 enumerates the transport types provided by the Memento infrastructure.

The primitive TType, *ReferenceDescriptor*, is important because it acts as the name of an MObject. An attribute with a ReferenceDescriptor value effectively points to the named MObject. It is through these kinds of attributes that groups of MObjects can be associated in more complicated constructs. The mechanism by which MObject names are resolved into MObject instances is discussed in Section 2.5.1.

Name	Description
Primitive:	
TInt32	32–bit integer
TUint32	32–bit unsigned integer
TInt16	16–bit integer value
TUint16	16–bit unsigned integer
TIndex	Small integer value
TBigIndex	Large integer value
TCount	Small cardinal value
TBigCount	Large cardinal value
TBool	Boolean value
TOperatingSystem	Enumeration value
TCoord	2–Dimensional coordinate value
TColorValue	Color intensity value
TDepth	Raster image depth
TString	Character string
TPoint	2–Dimensional coordinate
DateTime	Date and time representation
TimeSpan	Span between two DateTimes
Blob	Binary large object
RgbValue	Red–green–blue ColorValue triple
Colormap	Raster image color map of RgbValues
RasterImage	Raster image representation
ReferenceDescriptor	MObject name
Aggregate:	
TSet	Set of unduplicated transportables
TList	Unordered list of transportables
TRecord	Structured record of transportables
Special:	
TAny	Maximal TType
TUnknown	Minimal TType

Table 2.1 Memento transport types

2.1.3 MTypes are Represented as MObjects: A Self–Describing Type System

Each MType is represented in Memento as an MObject. The Memento type hierarchy forms a tree of such MObjects. This tree *is* the definitive representation of an assistant's type system. The Memento type system is therefore self–describing. The primary benefit of this direct representation is the ability to dynamically modify the MType system. Further benefits and implications are discussed in Section 2.8. All MType modifications are subject to the constraints over MTypes as described earlier. These constraints are part of the Memento transaction validity check over MType MObjects.

At the root of the hierarchy is the MType for all MObjects. This MObject instance is denoted as the `MObject` MType.[6] Since it is the root of the MType tree, it is the root of the MType hierarchy; all MObjects are members of the MType `MObject`. By implication, all MObjects must have the attributes required by the `MObject` MType. Table 2.2 shows the required attributes for all MObjects. The `MObject` MType has two required attributes: "modDate" and "mtype." A read–only MObject modification time, "modDate," is maintained by the Memento infrastructure. Additionally, the "mtype" attribute of an MObject points to the MType MObject of its MType. Changing an MObject's MType requires only changing the value of this attribute. Recall, however, that a transaction's validity checks will verify that the MObject has the attributes required of that MType by commit time. The default value of the "mtype" attribute points to the `MObject` MType, itself. By implication, the default MType is `MObject`.

Immediately under `MObject` in the Memento type hierarchy is the `MType` MType, or the MType of all MObjects in the type hierarchy. As stated earlier, MTypes are composed of a name, an optional supertype, a set of subtypes, a set of required

Name	TType	Default Value
modDate	DateTime	—
mtype	ReferenceDescriptor	`MObject`

Table 2.2 Required attributes of the `MObject` MType

[6] A monospace font is used for concrete MObject instances.

attributes, and a set of MObjects of exactly that MType. These five items take the form of required attributes on MType MObjects which are summarized in Table 2.3.

A few items in Table 2.3 deserve further comment. Optional items are represented as a *LIST* $[0 \ldots 1]$ of that item, implying that the list may have a cardinality between 0 and 1. A required attribute is represented as a record containing the fields "name," "ttype," and "default." The "name" entry is the name of the attribute, represented as a String. The "ttype" entry is a String representation of the attribute value's TType. Finally, the "default" entry is an optional default value for the required attribute which must conform to the TType in the "ttype" entry. The three entries in this record correspond to the three columns needed to enumerate the required attributes of an MType, such as Table 2.2 and Table 2.3.

The Memento type system is both self–describing and self–referential. All information about the type system exists in the MObjects representing MTypes. The `MType` and the `MObject` MObjects both have "mtype" attributes that point to the `MType` MObject. This is tantamount to saying that `MType` and `MObject` are both MTypes. The `MType` MObject describes itself and is self–referential in much the same way that Smalltalk's [51] *MetaClass* class is self–referential.

Transactions involving MTypes must undergo additional validity checks. The MType tree must remain singly–rooted at `MObject` and doubly–linked through its "supertype" and "subtypes" attributes. MType names must be globally unique. The name of any requirement must be unique along all possible paths through its MType to the root MType. The supplied default value for a requirement must conform to the requirement's TType. Every MObject of a particular MType must point to the MObject for that type. Finally, that MType must have exactly that set of MObjects as the value for its "mobjects" attribute.

The Memento type system can be dynamically modified. Two common modifications are to add a new MType and to add a requirement to an existing MType. To add a new MType, a new MObject must be created with the required attributes of `MType`. The new MType must be pasted into the MType tree at the appropriate place. The new MType must be added to "mobjects" attribute of the `MType` MObject,

Alternatively, to add a requirement to an existing MType, the "requirements" attribute of the MType must be appropriately modified: the set of requirements fetched, a record representing the new requirement added to the set, and the new set written back as the new attribute value. Note that in this situation, any MObject having that MType must participate in the transaction's commit–time validity check.

<table>
<tr><th>Name</th><th>TType</th><th>Default Value</th></tr>
<tr><td>name</td><td>String</td><td>—</td></tr>
<tr><td>supertype</td><td>LIST [0...1] of ReferenceDescriptor</td><td>Empty LIST of ReferenceDescriptor</td></tr>
<tr><td>subtypes</td><td>SET of ReferenceDescriptor</td><td>Empty SET of ReferenceDescriptor</td></tr>
<tr><td>requirements</td><td>SET of RECORD of:
<table><tr><th>Name</th><th>TType</th></tr><tr><td>name</td><td>String</td></tr><tr><td>ttype</td><td>String</td></tr><tr><td>default</td><td>LIST [0...1] of ANY</td></tr></table></td><td>Empty SET of RECORD of ...</td></tr>
<tr><td>mobjects</td><td>SET of ReferenceDescriptor</td><td>Empty SET of ReferenceDescriptor</td></tr>
</table>

Table 2.3 Required attributes of the `MType` MType

In particular, any such MObject that has an attribute whose value does not conform to the requirement's TType will cause the transaction to fail. In such a case, a transaction commit failure can be avoided by repairing errant MObjects as part of the transaction.

The bookkeeping details of nearly all common operations are handled by the Memento infrastructure itself. Assistant developers need not be aware of the minutiae described here.

2.1.4 A Small Example: A Consistent Set of MObjects

Figure 2.1 shows a Memento state with five MObjects. *mobject1* and *mobject2* are the `MObject` and `MType` MTypes, respectively. *mobject3* is a new MType describing cars. It has required attributes for model and year. *mobject4* is a generic MObject with a non–required attribute: “comment.” *mobject5* is an instance of the `Car` MType.

2.2 Accessing and Modifying MObjects: The Transaction Mechanism

An assistant's user interface is the primary source of MObject creation, modifications to MObjects, and MObject destruction. All such operations are done in the context of a Memento transaction. Like database transactions, Memento transactions have

mobject1: `MObject` MType	
modDate:	⋯
mtype:	*mobject2*
name:	"MObject"
supertype:	—
subtypes:	{*mobject2*, *mobject3*}
requirements:	{ [*name*: "modDate"; *ttype*: "DateTime"; *default*: ()], [*name*: "mtype"; *ttype*: "ReferenceDescriptor"; *default*: (*mobject1*)] }
mobjects:	{*mobject4*}

mobject2: `MType` MType	
modDate:	⋯
mtype:	*mobject2*
name:	"MType"
supertype:	*mobject1*
subtypes:	{}
requirements:	{ [*name*: "name"; *ttype*: "String"; *default*: ()], [*name*: "supertype"; *ttype*: "*LIST* [0 . . . 1] of ReferenceDescriptor"; *default*: (())], [*name*: "subtypes"; *ttype*: "*SET* of ReferenceDescriptor"; *default*: ({})], [*name*: "mobjects"; *ttype*: "*SET* of ReferenceDescriptor"; *default*: ({})], [*name*: "requirements"; *ttype*: "*SET* of *RECORD* of: [Name TType; name String; ttype String; default *LIST* [0 . . . 1] of TAny]"; *default*: ({})] }
mobjects:	{*mobject1*, *mobject2*, *mobject3*}

mobject3: `Car` MType	
modDate:	⋯
mtype:	*mobject2*
name:	"Car"
supertype:	*mobject1*
subtypes:	{}
requirements:	{ [*name*: "model"; *ttype*: "String"; *default*: ()], [*name*: "year"; *ttype*: "Count"; *default*: ()] }
mobjects:	{*mobject5*}

mobject4: An MObject	
modDate:	⋯
mtype:	—
comment:	"a note"

mobject5: Typical Car	
modDate:	⋯
mtype:	*mobject3*
model:	"Edsel"
year:	1956

Notes: Default values are indicated by a dash (—). Uninteresting values are elided (⋯). Sets, lists, and records are enclosed in curly brackets ({}), parenthesis, and square brackets ([]), respectively. Strings are quoted.

Figure 2.1 A consistent Memento state with five MObjects the traditional properties of *atomicity*, *consistency*, *isolation*, and *durability*. They are sometimes called the *ACID properties*.

Atomicity of a transaction means that if the transaction succeeds, all operations become permanent, otherwise there is no net effect. While all the operations in a Memento transaction take effect atomically, the actual set of operations performed may differ from those explicitly specified in the course of the transaction. In this respect, Memento's transaction mechanism relaxes the traditional property of atomicity. Memento's change notification mechanism, discussed in Section 2.3, is closely tied to the transaction mechanism. Together, these mechanisms make it possible for a software component that has registered interest in an MObject to react to notifications of trial changes to the MObject during the course of the transaction and, in turn, perform further operations on other MObjects that are part of the original transaction. Such spawned operations may ultimately affect the transaction's ability to successfully complete. The ability of a software component to modify MObjects in response to a change notification is described further in Section 2.3.4.

For a group of operations, the assistant explicitly begins a transaction, performs the operations, and attempts to *commit* the transaction, or make the affects of the transaction permanent. For simple operations, an assistant may rely on the fact that every MObject modification method or function will run a transaction for the sole purpose of carrying out the operation on the client's behalf, if one is not pending. During the course of a transaction, one of the operations may tag it as invalid—unable to commit. Invalidating a transaction is most often done when a complex operation cannot succeed and unwinding its effects from the rest of the operations in the transaction is impractical or impossible. Invalidation of a transaction may also be performed by a software component that has registered interest in an MObject via Memento's change notification mechanism.

If a transaction has not been invalidated, its success is determined at commit time by performing a validity check on all of the MObjects affected by the transaction. Specifically, the `isValid()` method is called on every `MObject` involved in the transaction. A negative result from any such MObject causes the transaction to fail to commit. It is through these validity checks that Memento's transaction mechanism adheres to the property of *consistency*, or the preservation of the integrity of its information.

Memento's transaction validity checking is extensible and customizable. Subclasses of `MObject` can add arbitrary constraints by overriding the `isValid()` method.

The `MObject` class's `isValid()` method always returns success. (An MObject's only constraints, those of its MType's attribute requirements, are maintained by Memento throughout all operations.) The `MType` class's `isValid()` method, however, verifies that the object belongs to a correctly specified MType tree. Further customization of these validity checks, described in Chapter 3, ensure that models built from MObjects adhere to the semantic constraints of an assistant's domain.

Isolation requires each transaction to observe a consistent database. The property prohibits one transaction from "seeing" the partially completed operations of other transactions. Within an assistant, this property is preserved by the fact that only one Memento transaction can be in effect at a given time. Currently, all Memento–based assistant's are single threaded. An assistant's transactions do not nest or overlap. This restriction is the result of a design decision for simplicity's sake. Research into transaction mechanisms for collaboration are beyond the scope of this dissertation. However, Memento has served as the vehicle for such research [68].

Collaboration among assistants is achieved via coordination through one or more shared MObject repositories. One assistant's transactions are isolated from those of other assistants through the mechanisms of transaction MObject read– and write–locking and *synchronization* which are part of the responsibilities of an MObject repository. Section 2.5 describes the concept of an MObject repository and details the interaction between a repository and the Memento transaction mechanism.

The property of transaction *durability*, or ensuring that a successfully committed transaction's effects are permanent, is outside the purview of Memento's transaction mechanism. This is because the transaction mechanism does not interact directly with a persistent store. The issue of durability is thus delegated to each Memento repository.

Transactions are managed by the single `Memento` class instance of each assistant. The `Memento` class provides methods for beginning, invalidating, committing, and aborting transactions as well as methods for testing the current transaction's state. As a convenience, most of these methods are also available as global functions that simply invoke the proper method of the single `Memento` instance.

The rest of the transaction mechanism operates through the methods and functions of the `MObject` class and its subclasses.[7] Memento's `MObject` class embodies

[7]In C++ parlance, *methods* are class operations on existing objects while *functions* are just procedures associated with the class.

the abstract MObject data type described in Section 2.1. Recall that in the data model, an MObject represents a set of name–value attributes. The model allows seven operations on MObjects: creation and destruction of an MObject, creation and destruction of an attribute, enumeration of an MObject's attributes, and access to and modification of the value of an attribute. For the sake of exposition, these operations can be divided into *accessors*, which retrieve information about an MObject and its attributes, and *modifiers*, which attempt to make a change. The next two sections describe accessors and modifiers provided by the `MObject` class and its subclasses and their relation to the transaction mechanism.

Occasionally an assistant will need to prevent an MObject from being modified for long spans of time. MObject read– and write–locks held during the course of a transaction are not appropriate for long–duration locking. When such a need arises, an assistant may lock an object via an *application–level* MObject lock. Only the assistant instance which holds an application–level lock on an MObject may modify it. Application–level locks are part of Memento's transaction mechanism and are discussed further in Section 2.2.3.

2.2.1 Accessing MObjects

`MObject` accessors are methods that get information about an MObject. All access methods in `MObject` subclasses derive from one of three access methods provided by the `MObject` class itself. The main one being `getAttributeValue()`. `MObject` subclasses typically provide their own, more useful access methods for attributes known to be required by the MType associated with the class. Such methods are called *convenience methods* as they just package the `MObject` access methods in a more convenient form. By convention, all access methods take an optional argument, called the `ReadFlavor` of the access, that indicates how the access will be performed relative to the current transaction. Because a pending transaction effectively maintains an alternate version of the MObjects involved, access methods must know which version to use. This section describes the three main `MObject` access methods, convenience access methods supplied by `MObject` subclasses, and the concept of a `ReadFlavor` that indicates how the MObject should be accessed with respect to any pending query.

The main `MObject` access method, `getAttributeValue()`, takes the name of an attribute and an optional `ReadFlavor` and returns a pointer to the (read–only) `Transportable` value of that attribute. If the MObject has no attribute by that name, a null pointer is returned. Recall that because the MType system can provide default values for required attributes, the access will return this default in case the attribute has not been overridden by the MObject. Less commonly used `MObject` access methods include a determination of whether an attribute value is overridden or defaulted and a request for all MObject attribute names.

When an attribute is known to be required, the `Transportable` pointer value returned by `getAttributeValue()` can be made more useful. Clients of the `MObject` class would like to retrieve a value of a more appropriate type from a method specific to the known required attribute. These so called *convenience methods* access the required attribute, downcast the resulting `Transportable` to the subclass associated with its required TType, perform any necessary type conversions, and return the value of the desired type. The primary responsibility of `MObject` subclasses is to provide convenience methods for their accessing and modifying their required attributes. The relationship between `MObject` subclasses and the MType system is described in Section 2.4.2.

The `MObject` class is strongly associated with the `MObject` MType, for example, which requires attributes for both the modification date and the `MObject`'s MType. The `MObject` class provides two convenience methods to get these values directly. The `getModificationDate()` method returns a DateTime representing the last modification date. The `getMType()` method returns the MObject's `MType` object which has been looked up using the required `ReferenceDescriptor` value. As with all `MObject` access methods, these methods take an optional `ReadFlavor` argument.

The `ReadFlavor` argument, which is required by all `MObject` subclass access methods, indicates how the access will be performed with respect to the current transaction. The `ReadFlavors` are *actual*, *filtered*, and *read–locked* with a *filtered* access being the default. An *actual* access gives a result that is independent of the state of the current transaction—the actual value. A *filtered* access on an MObject that has been modified by the current transaction gives results *as if the transaction were to succeed.* Any modification done in a pending transaction shows up in a filtered access. For example, newly created MObjects are potentially transitory, but they can be queried and modified as if they were real. If an MObject is not involved in a transaction or there is no pending transaction, a *filtered* access just returns the *actual* result.

The third `ReadFlavor`, a *read–locked* access, retrieves the most up–to–date value and prevents other assistants from changing the MObject until the transaction completes. Such accesses are useful when an assistant wants to read an attribute's value and then write a modified version of it back—for example, incrementing a counter. *Read–locked* accesses acquire a read–lock on the MObject which is held throughout the transaction. During the read–lock acquisition, attributes on the MObject may be updated to reflect any change to the MObject from where it is stored.[8] Any later modification to the MObject in the transaction will attempt to promote the read–lock to a write–lock prior to the write. This simple method of promotion of locks over the lifetime of a transaction gives the desired correctness properties while avoiding deadlocks.

2.2.2 Modifying MObjects

`MObject` modifiers are methods or functions that make modifications to MObjects. All such modifiers derive ultimately from two `MObject` methods and one `MObject` function. The `setAttributeValue()` method can create an attribute, change an attribute value, or remove the attribute altogether. The `destroy()` method destroys an existing MObject and the `createMObject()` function creates new ones. Like the `MObject` access methods, these methods and functions are customized extensively in convenience methods and functions provided by `MObject` subclasses. Also, as a convenience to assistant developers, all such methods and functions will create a transaction for the purpose of carrying out that single operation if one is not pending.

Modification of an `MObject`'s attributes is done through its `setAttributeValue()` method. The method requires the name of the attribute and a (possibly null) pointer to a Transportable value to set for that attribute. The method returns a boolean value indicating its success. The effect of passing a null pointer depends on whether the attribute is required by the MType system. If the attribute is not required, passing a null will remove the attribute. For a required attribute, passing a null will revert the attribute to its default value, if one is supplied by the MType's requirement for that attribute. Removing a required attribute without a default is not allowed; a failure result is returned. Other reasons for `setAttributeValue()` failure include trying to set a required attribute to a value that does not conform to the requirement's TType, trying to modify a read–only attribute, or failure to acquire a write–lock on the MObject. Note that none of these failures necessarily invalidates the transaction. MObject locking and storage are discussed further in Section 2.5.

[8]Because there is no convenient way to return failure from an access in the case of a read–lock failure, such an access returns the cached, actual value and tags the transaction as invalid.

As with MObject access methods, MObject subclasses provide convenience methods for simplifying the modification of required attribute values. A typical required attribute modification method will take a single argument of a type that makes sense to the method, convert it to the appropriate Transportable type, call setAttributeValue(), and return its result. In the case of the two required MObject attributes, the MObject class has a setMType() method to set the MObject's MType. Because the modification date attribute is read–only, no convenience function for setting its value is supplied. Such a method, were it to exist, would always fail. More complicated modification methods typically read an attribute value (via a read–locked access), and set new values based on supplied arguments in the context of a single transaction. One example is adding an element to a set.

The MObject class provides a function to create a new MObject in a specified repository where it is to be stored. Repositories are the subject of Section 2.5. This createMObject() function returns a pointer to the new MObject, if the specified repository will allow its creation. Failure is indicated by the return of a null pointer. The MObject returned by createMObject() is temporary over the course of the transaction. Only upon a successful commit of the transaction does the MObject become permanent. MObject subclasses supply similar functions to create MObjects of the appropriate MType. These creation convenience functions take additional arguments that are used to fill in the subclass–specific values for required attributes.

The final MObject modification method, destroy(), destroys the MObject. Again, this effect does not become permanent until the transaction commits. The destroy() method returns a value indicating its success. Subclasses of MObject do not usually override this method or provide related convenience methods.

By convention all modification methods and functions will begin a transaction on the client's behalf, if none is currently pending. The convention eases assistant developers' burden and makes the methods less reliant on the circumstances in which they are invoked. For the case of just setting a value, the setAttributeValue() method performs this service. Similarly, the createMObject() function and destroy() function provide this service. Modification methods that require multiple accesses to one or more MObjects must be able to run a complete transaction during the method. In cases where a modification method runs a transaction on the client's behalf, the method's return value indicates the success of the entire transaction.

2.2.3 Application–Level MObject Locking

Occasionally, an assistant will need to lock an MObject so that other assistant instances may not modify it. MObject locks acquired during transactions may only be held for the duration of the transaction, so they may not be used for this purpose. To address this need, Memento has a notion of *application–level MObject locks*. Such locks are implemented via a special, non–required, MObject attribute that is maintained by the Memento transaction mechanism. The presence of the attribute indicates that the MObject is locked. The value of the lock attribute indicates the assistant instance that holds the lock. An application–level lock may be acquired in one transaction, held for an arbitrary length of time, and unlocked in a later transaction. While the lock is held, only the assistant holding the lock may modify the MObject. Another mechanism, not discussed in this dissertation, ensures that an assistant's application–level locks are removed when its execution terminates. An application–level lock may only persist as long as the assistant instance that created it.

2.3 The MObject Change Notification Mechanism

The MObject change notification mechanism allows software components to receive notification of changes to MObjects. It is a customization of Memento's general change notification mechanism described in Section 2.3.1. The mechanism enables an assistant's user interface, for example, to react appropriately to modifications of displayed MObjects. User interface components typically register themselves as observers on MObjects that contribute to their display. When the component receives a change notification, it can determine the exact nature of the change and update the display accordingly. Because the change notification mechanism is invoked regardless of the source of the change, MObject updates in an assistant can be handled independently of how the change was instigated. This independence allows developers to make an assistant fully collaborative with little or no additional effort.

The MObject change notification is closely tied to the transaction mechanism described in Section 2.2. Recall that all modifications to `MObjects` occur during a transaction. MObject change notifications are dispatched both when the modification is requested and upon transaction commit. An `MObject` observer may therefore react to incremental changes (which would be undone in the case of a transaction abort) or permanent modifications to an `MObject`. In the spirit of the `MObject` subclasses' convenience methods, observers can receive notifications in a form that is more convenient to interpret. During a notification, an observer may react to the change by making further `MObject` modifications. The new modifications will become part of the pending transaction or begin a follow–on transaction, depending on the whether the reaction was to an incremental or permanent change notification. Sections 2.3.2 through 2.3.4 discuss these issues in more detail.

2.3.1 Memento's Generic Change Notification Mechanism

Memento's MObject change notification mechanism is built upon a more general change notification mechanism which is used whenever objects that are to react to changes need to be programmatically decoupled from the source of the change. Memento's generic change notification mechanism is based on that of the Smalltalk [51] system. Smalltalk's change notification mechanism is an integral part of its well known Model–View–Controller (MVC) user interface paradigm [71]. Within the paradigm, the model is an abstract representation of something to be manipulated, like a counter. The view provides a visual display of the model, perhaps a digital readout of the counter. The controller, usually intimately tied with a view, allows user input to be directed to the model. A controller might take the form of a menu associated with the view that increments and decrements the counter.

The main advantage of the MVC paradigm is that the model need not be aware of the number of views (or controllers) that rely on it or their nature. When the model changes in a significant way, it sends itself a message that describes the type of change that has been made in terms of an integer "part number." The counter might send itself a message that its value has somehow changed. As a result, all views are appropriately notified. Each view then asks the counter for the correct value and redisplays itself. In Smalltalk, any object can play the role of the model or the view, hence the protocol for both is part of the Smalltalk root `Object` class.

In Memento, two mix–in classes are used to play the role of change generator changes and change observer. These two classes are `Observee` and `Observer`, respectively. Instead of being available in all `Objects`, the mix–ins are blended into the inheritance hierarchy only as needed. As in Smalltalk, the `Observee` does not know the nature or number of its `Observers`, but it can notify them of changes by calling a change notification method. Instead of specifying a part number, however, an `Observer` creates a `Change` object to describe the change. The `Change` object can be queried for both the `Observee` that changed and a part number that is specific to that `Observee`. Not all changes are simple, however. Memento's change management mechanism allows for more information about a change to be encapsulated in `Change` subclasses. Such classes are usually intimately associated with a particular `Observee` subclass as they describe non–trivial changes to it. An `Observer` receiving such a change would know to downcast the `Change` object to its appropriate subclass based on the `Change` object's part number.

As in Smalltalk, Memento's change management mechanism is used extensively in the construction of an assistant's user interface. Typically, a Memento–based assistant has user interface components that correspond to MObjects representing models from the assistant's domain. Memento's `MObject` class inherits from `Observee`. User interface components in the assistant typically inherit from `Observer` and they watch for changes in various MObjects. Regardless of whether changes to models are made by the assistant itself or from external sources, such as other assistants, the user interface can react to the changes and update the user interface appropriately. The details of Memento's MObject change notification mechanism follow.

2.3.2 Notification During the Three Phases of a Transaction

Changes to `MObjects` are described by objects of the `MObjectChange` class, or one of its subclasses. The `MObjectChange` class extends the general `Change` class to include fields that enumerate the MObject being changed and when, during the course of a transaction, the change occurs. This field, called the `ChangeNotificationType`, can have one of three values: *trial*, *revert*, or *actual*. *Trial* change notifications are sent in response to every successful `MObject` modification during the course of a transaction. They are so named because they are temporary until the transaction successfully commits. If the commit attempt fails, a batch of notifications is sent to *revert* the effects of the previous trial changes. Finally, on a successful commit, a batch of notifications is sent to summarize the resulting *actual* changes to the MObjects. For both *actual* and *revert* change notifications, the order of the original MObject modifications is lost.

In the case of a repeated modification to an MObject's attribute value, one trial change will be given immediately after each modification, and either one revert change or one actual change will occur at commit time. Note that trial changes correspond to *filtered* accesses on MObjects and actual changes correspond to *actual* accesses.[9] MObject observers generally only react to notifications of actual changes, but the other notifications are useful in the user interface where incremental changes can provide the user immediate feedback on the transaction in progress. A user interface component might react to trial change notifications on an MObject's position attribute, for example, for keeping the display up–to–date while the MObject is being moved with the mouse.

2.3.3 Low– and Intermediate–Level Change Notifications

The MObject change notification mechanism dispatches general *low–level* notifications of MObject modifications and often, more detailed *intermediate–level* notifications.[10] *Low–level* notifications are in response to either a change in an MObject's attribute or the destruction of an MObject. Both change types are indicated by unique change part numbers associated with the `MObject` class. Attribute changes, which include attribute addition, attribute removal, and value modification, are described by objects of the `AttributeChange` class. The class inherits from `MObjectChange` and adds methods for accessing the name of the attribute, the nature of the change (addition, removal, or value modification), the old and new values (if they exist), and whether or not the old or new values override the default given by the MType system. Note that the `AttributeChange` class can only return information about the attribute value as a transportable, which is not convenient to manipulate. The destruction of an MObject is indicated by an object of the `MObjectChange` class as additional detail is not required. No notification is dispatched for the creation of MObjects as these objects cannot have observers at the time of creation. The MType system can be observed, instead, for notification of the creation of `MObjects` based on their `MTypes`. Section 2.4 describes an assistant's MType system in detail.

When they are given, *intermediate–level* MObject change notifications are dispatched immediately after the low–level notifications. Intermediate–level notifications, which are given only for required attribute value changes, mimic the convenience access and modification methods of `MObject` subclasses for required attributes. Intermediate–level notifications provide the same information to observers as low–

[9]Filtered and actual MObject accesses were described in Section 2.2.

[10]As one might expect, there are high–level notifications, as well. Such notifications are described in Chapter 3.

level notifications, but in a form that is more useful. Intermediate–level MObject change notifications fall into one of four categories: *simple*, associated with a set of ReferenceDescriptors, associated with a list of ReferenceDescriptors, or *special*. In the case of a *simple* intermediate–level notification, an `MObject` subclass associates a part number with an attribute required by its associated MType and provides an intermediate–level change notification of the `MObjectChange` class when that attribute changes value. An observer need only watch for a `Change` of that part number. For example, changes to the modification date, a required attribute of all MObjects, are given as simple intermediate–level change notifications.

Simple intermediate–level MObject change notifications are not adequate in all circumstances. When the attribute is a list or a set, for example, observers often want to know how the list changed or what was added or removed from the set. Recall that MObject attribute value modifications are atomic and such incremental changes are not saved for later notification. Instead, incremental changes are reconstructed based on the old and new sets or lists at notification time. Required attributes that are sets or lists of ReferenceDescriptors[11] are common enough to warrant their own special intermediate–level notification handling. Changes to sets of ReferenceDescriptors are described by objects of `RefSetChange` class. Here, two part numbers are associated with additions to or removals from the set. An intermediate–level notification is given for *each* addition or removal. Set additions and removals are reconstructed using the `TSet` class's set difference operator. The `RefSetChange` class inherits from `MObjectChange` and adds a field for the added or removed Reference looked up from its ReferenceDescriptor.

Similarly, changes to lists of ReferenceDescriptors are described by objects of the `RefListChange` class. A series of intermediate–level notifications indicate how the old list was edited into the new list with operations of insertion, deletion, or alteration of an item. The `RefListChange` class inherits from `MObjectChange` and indicates the type of edit, the old and new Reference(s) involved, and positional information from both lists. The "edits" to the list are reconstructed at notification time using the well–known minimum cost string editing algorithm embodied in the `TList` class. The `MObject` subclass supplies the costs and change part numbers associated with each of the three edit operations.

[11] Recall that ReferenceDescriptors are effectively pointers to other MObjects.

All other intermediate–level change notifications, called *special* notifications, are handled in an ad–hoc manner. Typically an `MObject` class will associate a specific `MObjectChange` subclass with a required attribute and a part number. The `MObjectChange` subclass provides special information about that particular modification. The required "mtype" attribute of the `MObject` class is one example of a special intermediate–level change. Notification of a change to an MObject's MType is described by an object of the `MTypeChange` class that enumerates the old and new MTypes.

Consistent with the general change notification mechanism, observers of low– or high–level MObject change notifications are only given a `Change` object for each notification that indicates the changed object and a change part number. Observers must do some work to extract the proper change information from this object. By convention, every `MObject` subclass exports a list of symbolic change part numbers unique to that class. The list always adds to the list exported by its superclass. Change part numbers are not globally unique, but they are unique along any inheritance path rooted at `MObject`. For this reason, an observer must determine the `MObject` of the change and its actual class before it can correctly discern the meaning of the change part value. Once the proper part is identified, the `Change` object can be cast to the `MObjectChange` subclass associated with the part for further interpretation.

2.3.4 MObject Modifications in Response to a Change Notification

While an assistant's user interface is the principal client of the MObject change notification mechanism, other software components can react to changes to `MObjects`. For example, Memento–based agents can watch for modifications in "interesting" `MObjects` and potentially perform changes to `MObjects` in response. Because Memento's transaction mechanism and the MObject change notification mechanisms are intimately tied, modification of an MObject in response to notification about a modification deserves further comment. There are three cases: modifications during a trial notification, an actual notification, or a revert notification. During a trial notification, the transaction of the source of the modification is still active and any further modifications become part of it. Recall that only one transaction can exist at a time. The success of the responding modification depends, therefore, on the success of the entire transaction.

Next, actual notifications resulting from a committed transaction effectively occur after the transaction has succeeded. Any `MObject` modifications will occur in their own, separate transaction. Many such follow–on transactions can be spawned from a single modification.

Finally, revert notifications, like actual notifications, occur after the failed transaction that initiated them. While it is possible to modify `MObjects` in response to a revert notification, doing so is not very useful. Most often, the goal is to undo whatever previous action was taken in response to earlier trial notifications. Because any such previous action becomes part of the (failed) transaction, its effects are erased by the same transaction failure, and so are the effects spawned by the previous action's notifications, and *their* effects, and so on.

With multiple agents generating various follow–on transactions, there is a potential for havoc. Infinite cascading transactions are possible with one or more agents manipulating the same sets of MObjects. Such agents have proved to be useful, however, as long as they are well behaved. The simplest case is an agent that maintains a constraint between two MObjects, say a and b. Such an agent would observe both objects and generate a modification of b in response to the proper change notification from a, and vice versa. If the agent's constraint between the two MObjects is *symmetrical*, an initial, external modification of a generates a response modification of b and the agent's response modification of b must be subsumed by the initial modification of a (and vice versa). Note that an attribute value modification to the same value sends out no notification, so the computation terminates in such an assistant regardless of whether a or b is modified first. The exact nature of a well behaved agent has yet to be fully characterized.

2.4 An Assistant's MType System

Each assistant maintains its own MType hierarchy which may change throughout its lifetime. This section discusses a number of issues relating to use and evolution of an assistant's MType hierarchy. Some of this material has been mentioned elsewhere. After describing how an assistant's MType system is represented by Memento, Section 2.4.2 describes how Memento maintains a correspondence between the MType hierarchy and subclasses of the `MObject` class used by an assistant. Section 2.4.3 describes how an assistant's MType system is initially constructed and evolves over the lifetime of an assistant instance.

2.4.1 Representation and Modification of an Assistant's MType System

The Memento type system, described abstractly in Section 2.1, is represented as a hierarchy of MObjects having type `MType`. Within an assistant, all such MObjects are represented by the `MType` class or one of its subclasses. The `MType` class inherits indirectly from the `MObject` class and adds convenience methods for accessing and modifying the attributes required of the `MType` MType. The Memento data model dictates several required attributes for MTypes: a name, an optional supertype (required except for the root of the MType hierarchy), a set of subtypes, a set of requirements on attributes that MObjects of the MType must contain, and the set of ReferenceDescriptors[12] of MObjects with that exact MType.

Practical considerations dictate that an MType's name and supertype attributes be read–only. A read–only MType name allows for a strong association between names and MTypes in the Memento code. A read–only supertype attribute forces the MType hierarchy to only be modified at its "edges" and not completely restructured. These two restrictions greatly simplify many programming issues and do not seem to place an undue burden on assistant developers.

Recall the primary reason for representing an assistant's MType system as MObjects is so that they could be readily modified and shared between assistants. Not only does each assistant have its own MType system, but most MObject repositories store their own MType system as well. When an assistant "connects" to a repository, the two MType systems must be reconciled in a process called MType *unification*. (A connection between two assistants is a special case of the repository MType system unification that is performed by the *Network Repository*, which is described in Section 2.5.6.) MType system unification, which is described in Section 2.4.3, is the usual source of dynamic changes to an assistant's MType hierarchy. However, it is possible and sometimes useful to directly modify the MType hierarchy.

One case in which it is useful to dynamically modify an assistant's MType hierarchy involves MIME types for "document objects." Document objects allow certain Memento–based assistants to facilitate collaboration over document files like those found in a file system. The MType for these documents is `DocumentConcept`. One of its required attributes is a Blob for holding the document's binary representation. An established mechanism for typing documents for transmission in electronic mail

[12]Recall that a ReferenceDescriptor is essentially a pointer to an MObject. ReferenceDescriptors and their relationship to MObjects is discussed in Section 2.5.1.

messages, called MIME, is in widespread use.[13] Rather than add a MIME type as a required attribute to every `DocumentConcept` MObject, MIME information is instead added once to the MType system. In particular, the `DocumentConcept` MType has a set of subtypes, one for each official MIME type. DocumentConcept MObjects do not have the `DocumentConcept` MType, but instead one of its subtypes—effectively associating MIME information with the document. Because each of these subtypes has attributes particular to MIME typing, it makes sense to require them in a new MType. This new MType, called the `DocumentMType` requires attributes for the MIME string, a human–readable description of the MIME type, and information about whether a file of that document type is stored as ASCII or binary. The `DocumentMType` MType is a subtype of `MType` (and is therefore a child of the `MType` MType in the MType hierarchy). By adding the MIME information to the MType hierarchy in this way, the set of official MIME types for a community can be stored in the same shared repository as the documents themselves. Maintenance of the set of DocumentMTypes is done by a simple administrative assistant. When the assistant modifies its MType hierarchy, those changes propagate out to its repository's MType hierarchy. Any assistant connected to the repository will "see" those MTypes added or modified by the same mechanisms as any other shared MObject.

While an assistant's MTypes are modified by the same mechanism as any other MObject, some of the transaction validity checks for MTypes are much more expensive than typical MObject modifications. For example, when a requirement is added to an MType, there is a potential for conflict between the requirement and an existing non–required attribute on some MObject of that MType. Recall that MObjects can have any number of non–required attributes of any TType. During a requirement addition or any TType change to an existing requirement, an exhaustive search must be performed on all MObjects of that MType for conflicting attributes. This search is performed during the validation phase of the transaction. Because the majority of the MObjects will not be in the assistant's memory, this search can be costly. Any such conflict will invalidate the transaction. Of course, such a transaction could be validated if it were to additionally remove or modify the offending attributes before commit time.

[13] "MIME" is an acronym for Multipurpose Internet Mail Extensions.

2.4.2 The Relation between an Assistant's MTypes and its MObject Subclasses

Within an assistant, there is a correspondence between MTypes and subclasses of the MObject class. This correspondence is often one–to–one, but it is possible that there are MTypes for which there is no MObject subclass. Subtyping of MObjects corresponds well to subclassing in object–oriented languages, like C++. Required attributes of the MType give rise to convenience access and modification methods and intermediate–level change notifications from the corresponding MObject subclass. The correspondence extends naturally into whatever MTypes and MObject subclasses arise from an assistant's domain.

The correspondence between MTypes and MObject subclasses in an assistant is represented and facilitated by objects of the MObjectClass class. For each MType (and MObject subclass) used by an assistant, an instance of the MObjectClass class characterizes the expected required attributes of the MType and the MType's expected relationship to others. The set of MObjectClass objects built by an assistant expresses the assistant's semantic constraints over the MType system. These objects assist in the original construction of an assistant's MType hierarchy and prevent undesirable modifications to it during the assistant's lifetime. The role of this class is described more fully below.

Once an MType and its expected required attributes are identified, creating the MObject subclass and MObjectClass instance are straightforward. Convenience access methods fetch the expected attribute by its expected name and down-cast the result to the expected TType. Similarly, convenience modification methods take objects of the expected TType and set them as attributes of the appropriate name. The MObjectClass associated with the MObject subclass is created via C++'s static construction (before assistant run–time). The MObjectClass constructor takes the information about the required attributes: the attribute's expected name, its expected TType, an optional suggested default value, and whether the attribute is expected to be read–only. Additionally, for each attribute, a change notification part number and information about the kind of change notification to be given for the attribute is also provided. Only if a special change notification is to be given is the developer required to override an MObject method to provide that notification. The various kinds of intermediate–level change notification types were discussed in Section 2.3.3.

Because Memento's MType system is dynamic, an MObject's MType may change during its lifetime. The `MObject` subclasses associated with an MObject's old and new MTypes may differ. To address this issue, Memento has a mechanism to *dynamically change the class* of an MObject. Such an operation is not supported by most object–oriented languages, including C++. Without this mechanism, however, a dynamic MType system would be largely useless to an assistant written in C++.

Memento changes an MObject's class through a fairly simple mechanism. Quite simply, the object is destroyed and reconstructed in place. Information associated with the old MObject, such as the MObject's overridden attributes and observers, later becomes associated with the new MObject. C++ allows the proper destruction of an object to be separated from its memory allocation. The memory of the old object is reused during the construction of the new one. By reusing the memory, any existing pointers to the object remain valid. Problems caused by different `MObject` subclass sizes are avoided because *all* MObjects are initially constructed with enough memory to hold the largest possible MObject. The maximum `MObject` subclass size is calculated just before the assistant's MType system is initially constructed.

2.4.3 Construction and Maintenance of an Assistant's MType System

An assistant's MType hierarchy is constructed from the information contained in its `MObjectClass` instances when the assistant begins execution. At that time, the MType hierarchy perfectly describes the MTypes associated with `MObject` subclasses in the assistant. Early in the assistant's lifetime, it connects to one or more repositories, described in Section 2.5. During a connection, the assistant's MType hierarchy may be modified based on the MType system associated with a repository. This process is called MType system *unification*. Throughout the lifetime of the assistant, its `MObjectClass` instances observe the MType hierarchy and react to its modification. The instances prohibit modifications to the MType hierarchy that might violate constraints that have been built into the assistant's code. Recall that it is the `MObjectClass` instances that capture the assistants semantic constraints over its MType system. This section describes the initial construction of an assistant's MType hierarchy and its maintenance during the lifetime of the assistant in more detail.

An assistant's MType system is constructed during the creation of the single, global `Memento` object with the help of the `MObjectClass` instances present in the assistant. The MType hierarchy is constructed by a single transaction. Each MType is created from information gathered by the corresponding `MObjectClass` instance. MType creation proceeds during depth–first traversal over the `MObjectClass` instances. As each MType is created, it is associated with the `MObjectClass` (and, hence the associated `MObject` subclass) during the remainder of the assistant's lifetime. This association is broken if the initial MType hierarchy construction transaction fails, or upon destruction of the `Memento` object when the assistant terminates.

Memento's MType system is dynamic; it may change during the lifetime of the assistant. Because an assistant contains code that may rely on certain features of the MType hierarchy, certain changes to the MType system may be damaging to the assistant. Most often, the features of the MType system relied upon by an assistant fall into one of three types of constraints. First, an assistant may require the existence of certain MTypes. For those MTypes, the assistant may rely on the existence of a set of required attributes with certain TTypes. Finally, the assistant may wish to limit MType changes on MObjects having particular MTypes. For all of these changes, the `MObjectClass` object associated with the MType contains the information necessary to check these constraints.

Recall that it is the `MObjectClass` instances that capture an assistant's semantic constraints over its MType system. During the time an `MObjectClass` object is associated with an assistant's MType, it observes changes to it via the MObject change notification that was described in Section 2.3. Upon the trial notifications of changes to the MType, the `MObjectClass` instance may invalidate the transaction or amend it so that the assistant's semantic constraints on the MType system are not violated. For constraints not fitting one of the above three categories, the assistant may observe the MType hierarchy directly and prohibit potentially damaging transactions in the same manner.

The most common source of MType system changes in an assistant is during the process of unification with a repository's MType system. Unification insures that the two MType systems are similar enough that MObjects can be shared between the assistant and the repository. The most common source of differences between an assistant's MType system and that of its repository is the evolution of the assistant's semantic constraints over time. The issue of evolution of assistants is discussed further in Section 2.8.

Unification proceeds by a comparison of the two MType hierarchies starting from their respective roots. Two MTypes may be unified if their names are the same and all attributes required by the assistant are present in the repository's MType. For any additional attribute required by the repository's MType, a default value must be provided; the assistant cannot provide one. If a required attribute of the assistant's MType is not part of the repository's MType, the repository's MType may be modified to include it. (Recall that this might be an expensive operation.) MTypes which exist in one MType hierarchy but not the other will effectively exist in the unified hierarchy. Finally, the unified hierarchy may not violate any of the constraints over the MType hierarchy outlined in Section 2.1.2 concerning attribute name duplication, and such.

2.5 MObject Storage

Memento stores its MObjects in *repositories* which act as an interface to a persistent store. Memento manipulates a repository and the MObjects in it via interfaces in the abstract classes `Repository` and `Reference`. The `Repository` class embodies an interface to a single store while objects in the `Reference` class act as handles to MObjects that might possibly reside in the store. When a developer creates a subclass of `Repository` for a particular store, he also creates a closely related companion `Reference` subclass.

A repository's responsibilities are similar to that of the entire first layer of the Memento infrastructure, but highly simplified. Whereas a transaction may span multiple repositories, a set of `Repository` and `Reference` subclasses concern themselves with the issues of a single store. A repository generally does not distinguish between MObjects of different MTypes in how they are stored or modified. A repository normally treats all MObjects as a simple collection of name–value attributes and takes advantage of the fact that the `Transportable` attribute values are serializable. While a repository is responsible for MObject read– and write–locking, it does not deal with the consequences of lock failure in the context of a transaction. Likewise, a repository must notice changes made to MObjects in its store, but it does not deal with the notification of those changes to an assistant's user interface. A repository has the additional responsibilities of persistently storing its MObjects; *realizing* its MObjects by creating an in–memory representation of them on demand; performing modifications to the stored representation of its MObjects; gathering modifications to stored MObjects made by other assistants using the same store; and management and storage of the MTypes of the MObjects stored therein. These responsibilities are discussed in detail in Section 2.5.1 through Section 2.5.5.

The most useful repositories can store arbitrary MObjects. However, a repository may instead act as a gateway to a legacy information system or store. The legacy store may not permit the full set of operations required by a general repository. Like a general repository, a legacy repository must translate an assistant's requests into operations that make sense in the context of the store. For example, it may only allow the storage of items whose mirrored MObjects conform to a restricted set of MTypes. Some repositories deal with legacy stores like CD–ROMs, which do not allow writes to existing MObjects or the creation of new ones. The general and legacy Memento repositories that have been implemented are described in Section 2.5.6.

2.5.1 MObject Names: References and ReferenceDescriptors

An object of the `Reference` class acts as a handle to an MObject that *might be* stored in a particular `Repository`. The set of all possible `References` associated with a `Repository` can be thought of as an abstract name space of MObjects in the repository. Because this set is infinite, objects of the `Reference` class are dispensed by its `Repository` only as they are needed or mentioned by a running assistant. A `Reference` object exists from the time it is mentioned until its `Repository` is destroyed—effectively the lifetime of the assistant. Therefore, clients of `Reference` objects need not be concerned about ownership of the object's memory or their premature deallocation. `Reference` objects have the property that any two specifications of the same MObject result in the same `Reference` object; no two `References` are associated with the same MObject.

MObjects would not be very useful if they could not have attributes which referred to other MObjects. Unfortunately, for programmatic reasons, the `Reference` class cannot inherit from the `Transportable` class. Another class, the `ReferenceDescriptor`, bridges the gap by acting as the transportable name of a `Reference`. The `ReferenceDescriptor` class inherits from `Transportable` and can be used as an MObject attribute value. A `ReferenceDescriptor` can be used to locate a `Reference` via a method in the single global `Memento` object. Conversely, the `Reference` class provides a method for returning its associated `ReferenceDescriptor`. `ReferenceDescriptor` objects are largely hidden from assistant developers. Convenience access and modification methods provided by the `MObject` class and its subclasses operate primarily on `Reference` objects and hide any necessary conversions to and from `ReferenceDescriptors`.

`ReferenceDescriptor` objects are themselves transportable, but they often need to be translated when they are transported between assistants or stored. For example, the network repository, described in Section 2.5.6, must perform `ReferenceDescriptor` translation between client and server. Care must also be exercised when constructing assistants that use multiple repositories. While it is useful and desirable to store `ReferenceDescriptors` in foreign repositories, such `ReferenceDescriptors` are rendered useless in an assistant that does not use the target repository.

2.5.2 MObject Realization and Access from a Repository

Each `Reference` object maintains a count of the number of software component users of its MObject within an assistant. The associated MObject is only guaranteed to be accessible if this count is non–zero. Any block of code that manipulates the MObject associated with a reference must therefore increment the count via the `increment()` method prior to accessing the MObject. After the access is complete, the count must be decremented via the `Reference` class's `decrement()` method.

When a `Reference` object's count is incremented from zero, it attempts to *realize* the MObject, or bring it into memory from the repository's store. Realization involves constructing an MObject of the appropriate `MObject` subclass and binding the MObject to the `Reference`. The choice of `MObject` subclass is based on the MType of the MObject and the correspondence between MTypes and `MObject` subclasses maintained by an assistant as described in Section 2.4.2. If the MObject cannot be realized, an exception is thrown out of the `increment()` method that must be caught by the code that attempted to increment the `Reference`. Causes for such a failure include prior destruction of the MObject in the repository or lack of appropriate access privileges. Likewise, when the Reference's count drops to zero, it may be purged from memory immediately or when the assistant requests unnecessary memory to be deallocated. A repository may elect to circumvent the default purge strategy and fix the MObject in memory throughout its lifetime.

While the `Reference` object's count is non–zero, it holds the MObject in memory. The `Reference` class provides a method to fetch a pointer to its associated MObject (or a null pointer, if the MObject is not realized). A convenient macro, `RefDerefSafeCast()` takes an `MObject` subclass *name* and a `Reference` and returns a pointer to an object of the appropriate class or a null pointer, depending on whether the `MObject` is in memory and has the requisite MType.

2.5.3 Repository Manipulation During a Transaction

The majority of methods provided by the `Repository` and `Reference` classes are manipulated "behind the scenes" by the `Transaction` class. The `Transaction` class, in turn, is manipulated primarily by the MObject class, as described in Section 2.2. As MObjects are manipulated during the course of a transaction, the repositories containing those MObjects are called upon to assist in the process. The following scenario demonstrates a repository's responsibilities in the context of a transaction. Typically, a read–lock is acquired on an MObject during a read–locked attribute access prior to its modification. During acquisition of the read–lock, the repository may update the in–memory representation of the MObject to accurately reflect the repository's current state. The read–lock is promoted to a write–lock upon the first modification of the MObject's attributes or when it is tagged for destruction. During the creation of a new MObject, the transaction asks the repository for a `Reference` that will be used for the MObject's storage. At commit time, the batch of changes on write–locked MObjects is checked to verify that the repository can perform the changes requested. When the transaction verifies that it can commit, the new MObjects are stored and the necessary modifications and destructions are performed on the MObjects involved in the transaction. Finally, regardless of whether or not the transaction committed, all locks are released. Where possible, repository operations are batched so that a particular repository can take advantage of less costly bulk operations. For example, at the end of a transaction, all read– and write–locks are released by a single method call.

An important responsibility of a repository is to notice changes to MObjects inside its store. It may notice these changes during any repository call, but it must explicitly gather all such changes when the repository's `synchronize()` call is made. *Synchronization* is described in the next section. Additionally, when a read– or write–lock is first acquired on an MObject, any update to that MObject must be immediately noticed. If a transaction sees such an update during the acquisition of a lock, the entire transaction becomes invalidated. Unfortunately, the update to a single MObject may need to be coordinated with updates to other MObjects from the repository to be meaningful. In such a situation, an assistant may retry the transaction after *synchronization*.

2.5.4 Synchronization: Transactions Emanating from a Repository

MObject modifications in an assistant are normally performed during a user–initiated transaction via the user interface. Periodically, however, the assistant must perform a *synchronization*. Synchronization allows changes to MObjects made by other assistants (and other means) to be incorporated into the assistant's in–memory copies of those MObjects. Because Memento is single–threaded and only allows one transaction at a time, synchronization is performed during times when the user interface is idle. To perform a synchronization, an assistant need only call the `synchronize()` method of the single `Memento` object. The `Memento` class provides a method that summarizes under what conditions the `synchronize()` method might perform useful work. The summary is gathered from a similar query on each repository. The assistant may use this summary in deciding a synchronization strategy that does not waste computing resources by polling unnecessarily.

A synchronization involves forcing each `Repository` to check its store for MObject modifications and deletions.[14] All such modifications are then performed in batch by a single transaction. The transaction updates the in–memory MObjects while avoiding unnecessary work in the repositories involved. Typically, a repository performs no locking and no modifications to MObjects' stored representations during a synchronization. Because MObject modifications can be spawned during the trial notification of the synchronization transaction, however, spawned modifications must be allowed to propagate back to their appropriate stores. In this case, the normal locking and modifications are performed only on the MObjects whose stored representations require modification. In effect, the synchronization transaction is "filtered" through the pending updates from the repositories. Without this great attention to avoid unnecessary locking and updates, a systemic performance loss would result.

There is a potential for the synchronization transaction to fail to commit. On transaction failure, several attempts are made to re-synchronize and re-commit. If

[14]MObject creations are not explicitly noticed during synchronization. All MObject creations involve a corresponding modification to the repository's MType system. Additionally, an MObject creation is often accompanied by the modification of MObjects that refer to the newly created MObject. These changes are noticed during the synchronization process.

the synchronization process ultimately fails, the assistant can no longer perform meaningful computation and should gracefully terminate. There are many situations under which termination of the assistant is the best course of action. Consider the case of a modification to a repository's MType system that would violate an assistant's semantic constraints. Upon synchronization, the repository may spawn modifications to the assistant's MType system (as described in Section 2.5.5). These changes are monitored by the `MObjectClass` instances that capture semantic constraints on the assistant's MType system. Any transaction that performs a modification that is incompatible to an assistant's MType system is invalidated by one or more `MObjectClass` objects. In such a case, synchronization would ultimately fail; the assistant should no longer continue execution as it is no longer compatible with the repository's MType system.

2.5.5 Repository–Specific MType Systems

Each repository must store its own MType system and hence be self–contained. Without such self–containment, a repository could easily get out of date with evolving assistants. When an assistant begins using a repository, the repository's and assistant's MType systems must be reconciled—a process called *MType unification*. MType unification usually involves making minor changes to a repository's MType system to make it correspond to the assistant's MType system. Typically, the addition of required attributes or new MTypes is all that is required to unify a repository's MType system with an assistant that has been improved since its last unification with the repository. The procedure used to unify MType systems was discussed in Section 2.4.3.

There are two strategies for storage and unification of MTypes from a repository. For repositories that can store general MObjects, it is generally most convenient to store the repository's MType system as a tree of MObjects. It is usually sufficient for such a repository to only know where it's root MType is stored. The `Repository` class provides routines for realization of the MType tree, unification with an assistant's MType system, and maintenance of the repository's MType system over the life of the assistant. The caller of the unification routines has the option of forcing the repository to match the relevant portions of the assistant's MType tree or abort the unification. It is usually desirable to have the repository's MType system be updated to reflect that of the assistant as it is the assistant's evolution that drives changes to the repository's MType system. Additionally, methods provided by the `Repository` class watch for the creation and destruction of MObjects in the repository and will extend or prune the repository's MType system as needed to have exactly the MTypes necessary to describe the stored MObjects.

In the second case, where a repository is a gateway to a legacy information system, MType storage and unification is handled a different way. Such repositories essentially have a fixed MType system as the legacy information system generally imposes restrictions on what can be stored. Generally, these repositories provide their own `MObject` subclasses for the MTypes exported by the repository. Through the MType system creation mechanism described in Section 2.4.2, these supplied `MObject` subclasses build their required MTypes into the assistant's MType system. The net effect is that the MTypes of objects "stored" in the Repository are hard–wired into the assistant. No real unification is required in this case. The gateway repository still bears the burden of updating the assistant's MType system with any new or destroyed MObjects of the various MTypes.

2.5.6 Implemented Memento Repositories

Memento currently has five repositories. More will likely be added as Memento matures. Two repositories, the *file repository* and the *network repository* can store arbitrary MObjects. A third, the *system repository* is special in that its MObjects only exist during the lifetime of an assistant. Two other repositories, the *Photo CD repository* and the *WWW (World–Wide Web) repository* are gateways to legacy stores. Of the five, the network repository is the most important as it allows the construction of assistants that facilitate collaboration over the Internet via UNIX–style sockets. Each of the five repositories is now described in turn.

The File Repository

The file repository, embodied in the `FileRepository` and `FileReference` classes, allows arbitrary MObjects to be stored as a group of files contained in a single file system directory. An assistant using the file repository may run anywhere it has access to the directory—either on the same machine or via an NFS (Network File System) mounted file system. Multiple assistants may each have file repositories on the same directory (store) and effectively collaborate through it. Alternatively, a single assistant will use as many `FileRepository` objects as it has file repository stores, although one is typical. The repository's portable file format is largely dictated by serialization methods provided by `Transportable` subclasses. In order to avoid problems caused by different file system permissions, each assistant must, unfortunately, use the same file system user identification throughout the FileRepository's lifetime. While this restriction may seem severe, it is not a problem in practice as file repositories are most often used by single users—either as a personal or debugging store, or as a dedicated server that shares its MObjects via the network repository, described below.

Within a file repository's directory, MObjects are stored, one per file, in files with numerical names. Within each file, the overridden attributes of the MObject are stored as a set of triples: the attribute name, the attribute value's TType, and the attribute value itself. Any modification to an MObject results in the entire file being rewritten.[15] The MObject's file's time stamp effectively indicates to the repository when the MObject was last modified.

The file repository maintains read– and write–locks as simple files with names related to the file name of the MObject being locked. The write–lock is a zero byte file that exists only if a write lock is held on the MObject. The read–lock file, if it exists, contains a non–zero count of the number of read–lockers on the MObject. A missing read–lock file implies no read–lockers. A file repository will modify the various MObject locks only when a write lock is held on the single, zero byte repository lock file. Access to the repository lock is done via "exclusive create" file locking that acts as a semaphore for multiple competing file repositories. The repository lock is held only during MObject lock manipulation, so it is quickly released and does not impose a bottleneck when multiple assistants are using the same file repository directory (store).

Several files in a file repository's directory keep information for the repository itself. The repository's internal name is stored in a single file. Another file stores an integer that is used as the name of the next MObject file to be stored in the directory. As with MObject locks, modification to this file occurs only while the repository's lock is held. Finally, a zero byte time stamp file is "touched" by a file repository after it modifies MObjects in the directory. The file's purpose is to accelerate the file repository's synchronization process. Upon synchronization, a file repository first

[15]An alternative strategy was tried where each overridden attribute of an MObject was stored in a separate file in a directory associated with the MObject. With a large number of small files, the cost of file opening and access became too large.

checks the modification time of this file. In the most common case where nothing in the store has been modified, the modification time of the time stamp file will not have changed and no further action is required. If the time stamp has been modified, an exhaustive search for modified or deleted MObject files is performed for MObjects in the assistant that have been realized from the repository.

The file repository maintains an MType hierarchy in the repository's directory for the MTypes of every stored MObject. The MType MObjects are stored in the same manner as other MObjects, however the `MObject` (root) MType is always stored as file "1". (A minimal repository MType system is constructed when the file repository's directory is first initialized.) Upon construction of a `FileRepository` object, the entire repository MType hierarchy is realized and unified with the assistant's MType system. An assistant may request that the file repository's MType system be modified to match that of the assistant, if possible. If unification is not possible, an exception is thrown out of the `FileRepository` object's constructor. During the lifetime of the `FileRepository` object, it relies on the methods of its `Repository` superclass to help it maintain the file repository's MType hierarchy. For example, if an MObject is to be created in the repository of an MType that the repository does not yet contain, the `Repository` class will extend the repository's MType hierarchy to include that MType. Extension of the repository's MType hierarchy may also occur if an existing MObject's MType is changed to one that the repository does not yet contain. Likewise, the `Repository` class may prune a repository's MType hierarchy during MObject destructions or MType changes. Unused MTypes are pruned from the file repository's MType hierarchy so that obsoleted MTypes do not impede future unifications.

The file repository can be used in two different ways, either *cooperatively* or *non–cooperatively*. When file repositories are run cooperatively, multiple assistants can collaborate via a single file system directory. There are many situations, however, which do not require the full generality of a cooperative file repository. For example, an assistant may use a file repository to store only MObjects that will not be involved in collaboration. Alternatively, a dedicated "server" assistant can store its MObjects in a file repository and export them for collaboration via the network repository, described later. When it is acceptable for only one assistant to use a file repository at a time, it may be used non–cooperatively with greatly reduced overhead. On start-up, a non–cooperative file repository acquires the repository lock and holds it for the lifetime of the assistant. Since no other assistant can access MObjects from the repository, there is no need for MObject locking or file system polling during synchronization. If an assistant's non–cooperative file repository is run with other cooperative ones, the cooperative assistants will be blocked until the non–cooperative assistant terminates. Other than that, there is no harm in mixing cooperative and non–cooperative file repositories on the same store.

The Network Repository

The network repository acts as a one–way connection between a single assistant that plays the role of a server and many other assistants which play the role of clients. The server assistant and its client assistants communicate via a single UNIX–style socket. The connection is one–way because MObjects from the server are visible and accessible via the clients, but client MObjects are not visible to the server. A server accepts connections on a known port via the UNIX system calls `bind()`, `listen()`, and `accept()`. Clients connect to the port via the UNIX `connect()` system call. Like with the file repository, the communication protocol is largely dictated by the serialization methods provided by `Transportable` subclasses. The client assistant then unifies its MType system with that of the server in the same way the file repository unifies the stored MType system with that of its assistant.

The network repository allows considerably more flexibility than the traditional client–server scenario where a single assistant, s, plays the role of a server to multiple client assistants, $c_1, c_2, c_3, \ldots, c_n$. Assistants only play the *role* of a client or server with respect to a single connection. Many variations are possible, however, as each assistant can have multiple connections and play different roles with respect to each of them. For example, the server, s can be the client of another assistant, t playing the role of server to it. Here, s sees t's MObjects and the original clients see MObjects from both. In another variation, two assistants can each play server and client to the other via two socket connections in a peer–to–peer relationship. MObjects from both assistants are visible to both assistants. The network repository prevents an infinite mirroring effect where an assistant would see its own objects that were exported to the other assistant and imported again. The implications of the network repository as a flexible Memento software component is discussed in Section 2.8.

Conceptually, the network repository is the simplest of the repositories because there is no translation to be done between the repository and its store. The concepts of locking, MObject realization, MObject modification, and MType system manipulation are an exact match between the client (repository) and server (store). Synchronization is straightforwardly implemented at the server with a `Watcher` class that gathers incremental changes in MObject attributes via the MObject change notification mechanism described in Section 2.3. Transmitting of MObject attribute values, for realization, modifications, and synchronization is handled by the transport mechanism as part of the `Transportable` class and its subclasses. Currently, the network repository server runs requests for its clients in single steps. A request can include a realization, a release of interest, a synchronization, or an entire client transaction. Future versions of the server code will allow multiple client transactions to be run concurrently in separate threads.

The network repository is generally more efficient than the file repository since only modified attributes are transmitted during modification or synchronization. File system polling is replaced by a blocking `select()` call on one or more socket descriptors. A typical dedicated server assistant will store its MObjects in a file repository. To avoid the latency of a file system fetch in addition to the required network transfer for an MObject's realization, such a server can easily be adapted to realize all of its stored MObjects at start time.

The System Repository

The system repository, embodied in the `SystemRepository` and `SystemReference` classes, is a general MObject repository, though it does not provide truly persistent storage of its MObjects. The system repository is intended to store MObjects that are to persist only for the lifetime of the assistant. The repository cannot realize MObjects, but it allows the creation of MObjects in it. MObjects in the system repository can be read– and write–locked without restriction (or work). Finally, since there is no store, no synchronization is required.

Every Memento–based assistant has exactly one system repository which is created by the single `Memento` object. Immediately after creating the system repository, the `Memento` object builds the assistant's MType system within it. The MType system was discussed in Section 2.4. The construction of the MType system was described in Section 2.4.2. Other assistant–specific MObjects, described in Chapter 3, are additionally created in the system repository at start–up time.

The Photo CD Repository

The Photo CD repository acts as a gateway to a Kodak photographic compact disk that stores multiple photographic images at a range of resolutions. The repository exports five different MObjects corresponding to the five MTypes that describe information from the repository. Most important are the MTypes corresponding to a physical compact disk and the `ImageConcept` MType that represents a photographic image at a specified resolution. While these MObjects can be stored in any repository, the photo CD repository itself is read–only. Realization of a photo CD's MObjects requires reading the appropriate information from the CD via a proprietary library purchased from Kodak. If the appropriate disk is not loaded in the CD–ROM drive when a realization request is made, the repository has the ability to ask the user to switch CDs on its behalf.

The WWW Repository

The World–Wide Web or WWW repository treats the Internet as a store of MObjects. WWW repository references are based on universal resource locators (URLs). Realization of the MObject amounts to fetching the resource published at the URL and realizing an MObject to hold it. Currently, the WWW repository can only fetch three kinds of pre–defined MObjects: images, text, and HTML documents. If the result of the fetch is an image, an `ImageConcept` MObject is realized with an "image" attribute containing the image. Similarly, an ASCII text result will cause the realization of a `TextConcept` MObject with a "text" attribute containing the text as a `Blob`. Finally, fetching an HTML document will result in its parsing. The resulting parse tree becomes an attribute on an `HTMLConcept` MObject.[16]

The WWW repository is not a general repository of MObjects. First, it is read–only as the WWW protocols for Gopher and HTTP are intended primarily for publication. Further, the repository only allows the fetching of MObjects that have close counterparts on the Internet. Finally, there is not a good mechanism for dealing with modification of the underlying network–based information, so synchronization is not attempted. The WWW repository was implemented as an experiment.

[16] The reason for the choice of "Concept" in these names will become apparent in the next chapter.

2.6 The Running of a Typical Collaborative Assistant

A typical run of a collaborative assistant involves all of the mechanisms described in this chapter. This section gives the reader a sense of an assistant's thread of execution while, at the same time, summarizing the various mechanisms from the first layer of the Memento infrastructure. A few remaining components are discussed here, as well. Start–up of the assistant, in particular, is quite complicated and deserves further comment.

In C++, so called statically initialized objects can be constructed *before* the `main()` function is called. Many Memento objects are constructed during this interval before "run–time." In particular, all primitive `TType` objects are constructed at this time. (Compound `TType` objects are created as they are demanded.) All `MObjectClass` objects which describe semantic constraints over the assistant's MType system are also constructed before run–time. This class was described in Section 2.4.2.

During the assistant's `main()` function, the single global `Memento` instance is usually the first object to be created. It must exist throughout the lifetime of the assistant. The single `Memento` object, which has only been briefly mentioned elsewhere, is the hub of many of the Memento services. In particular, the `Memento` object:

- creates and maintains the `SystemRepository`, described in Section 2.5.6;
- coordinates the construction of the assistant's MType hierarchy described in Section 2.4.2;
- maintains a registry of existing repositories and answers queries for `Reference` objects from those repositories, as described in Section 2.5.1;
- provides a hub for the starting, committing, and aborting of transactions, described in Section 2.2;
- provides methods to be overridden by `Memento` subclasses to give user feedback during prolonged operations, such as changing the mouse cursor; and
- provides methods for repository synchronization, described in Section 2.5.4, which must be called periodically by the assistant.

After constructing the `Memento` object, an assistant typically interprets its command line options and connects to one or more repositories. Unification with a repository's MType system, described in Section 2.4.3, may result in modifications to the assistant's MType hierarchy. All such modifications are observed by the assistant's `MObjectClass` instances (Section 2.4.3) to ensure that the assistant's semantic constraints over its MType system are not violated. The `MObjectClass` instances may invalidate or amend unification transactions or any other transactions involving the assistant's MType hierarchy. A failure to unify the repository's MType system usually results in an exception to be thrown out of the particular repository object's constructor. Existing repositories were enumerated in Section 2.5.6.

After the assistant's repositories have been created, the assistant realizes MObjects from the repositories that are of interest to it. MObject realization and access were the subject of Section 2.5.2. An assistant will realize MObjects that comprise models from its domain. How an assistant finds the models in its domain is discussed in Chapter 3.

Typically, an assistant's user interface will display the information contained in one or more MObjects. User interface components act as observers on those MObjects and receive notifications of changes to them. The MObject change notification mechanism was the subject of Section 2.3. Because notifications do not differentiate between MObject modifications that are initiated by the assistant itself or by repository synchronization, assistants may be made fully collaborative with little change to the assistant's code.

An assistant's user interface is the primary source of modifications to MObjects, their creation, and destruction. All such operations are done in the context of a transaction, the subject of Section 2.2. If successful, the transaction results in the modification of the MObjects' representations in their persistent store. This manipulation is facilitated by a `Repository` object that acts as an interface to the store. Repositories and their responsibilities were the subject of Section 2.5.

When the assistant's user interface is idle, it must periodically perform a synchronization operation. The operation, discussed in Section 2.5.4, allows changes to MObjects' representations in a persistent store to be integrated into the `MObject` objects realized in the assistant. It is through this synchronization operation that MObject modifications made by other assistants find their way into an assistant. Synchronization is an important part of how the lower layer of Memento facilitates collaboration.

Shutdown of the assistant proceeds in roughly the reverse order as the start–up. First, all MObjects that have been realized and observed by the assistant must be released. Failure to fully complete this step is the primary source of shutdown failures in assistants. Any repositories created by the assistant must then be destroyed. Finally, the `Memento` object itself must be destroyed. After exit of the `main()` function, the various statically created objects are then automatically destroyed by the C++ language's runtime support.

2.7 Relation of Memento's First Layer to Other Technologies

Memento's first layer has much in common with toolkits that support artifact based collaboration, object–oriented databases, and middleware infrastructures. Memento's relation to artifact–based collaboration toolkits is explored fully in Section 5.1. Memento's relation to the other two technologies is now discussed.

Object–Oriented Databases

Memento's first level, as presented in this chapter, addresses many of the same issues as object–oriented databases (OODBs). Both have an object–based data model. Both deal with persistence of objects, inheritance hierarchies, coherent transactions, and schema evolution [21]. In order to explore the relationship between OODBs and Memento's first layer, three scenarios are explored:

- replace Memento's first layer with an OODB,
- claim that Memento's first layer is an OODB, or
- use an OODB as a specific MObject repository.

Of the three scenarios, only the last one is feasible.

There are two reasons why an object–oriented database could not replace Memento's first layer, as suggested in the first scenario. First, an important practical feature of the Memento infrastructure is the abstraction of the MObject repository. Memento allows MObjects from various repositories to participate in transactions independent of their origin. If Memento were limited to a single persistent store, such as that of an OODB, Memento–based assistants would loose a considerable amount of configuration flexibility. A distributed OODB might well replace the network repository when it is used as a centralized server, but such a database could not replace the network repository when used in a peer–to–peer configuration. Similarly, replacing Memento's first layer with an OODB would preclude the use of repositories connecting to legacy stores such as the Photo CD Repository and the WWW Repository. Memento's design gains greatly by the separation of its transaction and evolution mechanisms from the mechanisms for the persistent storage of MObjects.

The second reason that Memento's first layer could not be replaced by an object–oriented database is that most OODBs lack the features necessary to support collaboration and dynamic modification of an object's class. Memento's collaboration capabilities might be replaced through an OODB's support for dynamic change or assisted by an OODB's support for cooperative transactions. However, such features are supported by only a few of the commercial OODBs [5]. A typical OODB relies heavily on object versioning for concurrency resolution and leaves the onus on the developer to resolve conflicts. With a typical OODB feature set, Memento would be required to poll the database for changed MObjects and merge versions, as necessary. Additionally, Memento's inherently dynamic type system is at odds with the type systems of most OODBs. In particular, few OODBs allow the dynamic changing of an object's class. This mechanism is critical for Memento's support of information space evolution, described in Section 3.6.3.

The second scenario suggests that Memento might be considered an OODB. Despite its many similarities to OODBs, however, Memento lacks many important features that an OODB should have. Even with the current lack of OODB feature set standards, it seems reasonable to expect that an OODB should support compound objects, long duration transactions, nested transactions, crash recovery, and query handling. OODBs are geared toward applications that modify large portions of the stored object set over a potentially long run of the application. Such assumptions are fundamentally at odds with Memento's design.

Finally, as the third scenario suggests, an OODB might be a good candidate for an MObject repository. In this scenario, the problems of collaboration support, mentioned earlier, would still need to be resolved. During a repository's synchronization process, described in Section 2.5.4, a repository must update the assistant's version of stored MObjects in the repository that have been updated since the last synchronization. Ideally, an OODB–based MObject repository would make use of change notification or cooperative transactions during synchronization. Such features, however, are not available in all OODBs.

Middleware

A new trend in software infrastructures, called *middleware*, allows the coherent integration of heterogeneous components. Middleware infrastructures focus on the issues of reduction of complexity and facilitating interoperability. Additionally, these infrastructures often provide meta–level services and other value–added facilities. Recent middleware infrastructures have tended to address one of three areas: the user interface, network distribution of information, and heterogeneous data repositories. The latter two are really just two different perspectives on the same problem of sharing information. The user interface middleware infrastructure, OpenDoc [87], allows a variety of small applications to work together in the context of a single document. CORBA [86] is a distributed object middleware standard that allows for the integration of a diversity of cooperating network–based information producers and consumers. IBM Almaden's ongoing Garlic [18] project attempts to integrate a variety of databases and other information stores to create a unified multimedia information system.

Memento, too, could be considered a middleware infrastructure for the integration of divergent information stores. Through Memento's repository concept, diverse information stores are coherently unified. Memento's `Repository` and `Reference` classes provide an abstract interface to an MObject repository that can be customized for a particular store. As Section 2.5.6 demonstrated, a variety of information stores can be unified through this interface.

2.8 Implications of Memento's First Layer

The design of the Memento infrastructure's first layer has some important consequences. The layer provides support for artifact–based collaboration over a simple, but general data type, the MObject. Memento facilitates the customization of MObjects for a particular assistant's needs, in part through its dynamic, extensible, self–describing MType system. Memento takes advantage of an explicit meta–data representation, which includes the MType system, to facilitate the evolution of assistants and their stored MObjects. Finally, Memento's software components can be combined in various ways that enable a wide variety of (assistant) applications.

Support for Artifact–Based Collaboration

The first level of Memento provides the mechanisms required to support artifact–based collaboration over MObjects. Repositories provide a persistent store for MObjects which is accessible by multiple (collaborative) assistant applications. Each transaction an assistant makes on an MObject effectively modifies the MObject's representation in its store. A repository's synchronization mechanism, combined with Memento's transaction mechanism, assures that each assistant that has in–memory copies of an MObject will be updated in a timely fashion. Regardless of how an MObject's in–memory representation is updated, Memento's notification mechanism allows interested user interface components to keep the displayed representation of MObjects current.

Generality of the Data Model

Memento's data model allows for the expression of a wide variety of assistant–specific data objects via a single representation—the MObject. Through `MObject` subclasses and by extension of the dynamic MType system, an assistant developer can easily customize the MObject data model for the specific needs of an assistant. Once the required attributes of an assistant–specific MObject have been determined, it is a simple matter to write the convenience access and modification methods to fetch and manipulate those attributes. The association of an assistant's `MObject` subclasses with their corresponding MTypes and the semantic constraints over an assistant's MType system are easily expressed in a declarative fashion by the assistant developer. Additionally, the most common change notification types can be declaratively specified at the same time.

Evolution of Assistants and their MObjects

Memento's design acknowledges that all applications undergo change. While many changes are cosmetic in nature, an important class of changes involves changes to the underlying data types. Such changes are especially troublesome in traditional artifact–based collaborative systems where *all* assistants and any shared artifact stores must all be updated simultaneously. This single constraint makes the development and evolution of collaborative applications painful at best without the aid of the Memento infrastructure.

Memento provides mechanisms for the graceful evolution of both assistants and the (MObject) artifacts stored in shared repositories. An assistant's expectations of its data model, the so–called semantic constraints over its type system, are easily expressed and modified by an assistant's developers. Both assistants and Memento MObject repositories have their own MType systems that describe the MObjects contained therein. Newly added MTypes and additional MType requirements can effectively flow from the new assistants to old assistants via their shared repositories. Old assistants can usually coexist with new assistants with the mechanisms provided by Memento. The entire system of collaborating assistants and their shared repositories of MObject can thus undergo a graceful evolution.

Flexibility of the Memento Architecture

Memento is a library of classes that can be used in a variety of ways. While Memento provides many platform–independent mechanisms, it does not impose policy decisions as to their usage. Memento components can thus be combined in a variety of ways. Of particular interest is an assistant's user interface and its use of repositories.

Memento places little restriction on the kind of user interface an assistant must have. Memento's change notification mechanism provides a very general way for the user interface or any registered software component to react to MObject changes. The look and feel of an assistant's user interface is completely up to its developers. Indeed, interesting assistants, such as information agents, need not have a user interface at all. The Memento infrastructure has been successfully combined with several application frameworks in the making of various assistants. A discussion of specific Memento–based assistants is left for Chapter 4.

Most of Memento's flexibility, however, comes from its repositories. Memento's file repository can be used to build a simple "modeler" which allows a user to work with data objects and store them between uses. By changing just a few lines of code, the same assistant can instead use a network repository that connects to a centralized assistant–independent MObject server. The resulting application would be a fully functioning collaborative application. The server can be augmented with special repositories of its own, perhaps as gateways to legacy stores, as was demonstrated by the Photo CD repository. A local server shared over a local area network can act as a cache for MObjects that reside in a globally shared server at a greater distance. Such a cache would effectively consolidate network traffic between it and the main server.

Chapter 3

Memento's Layer II: Abstraction of Meaning

The previous chapter described the layer of the Memento infrastructure that supports artifact–based collaboration over general MObjects. This chapter will describe Memento's mechanisms to address the semantic needs of assistants which were outlined in Chapter 1. These mechanisms, which build on the foundation of Memento's first layer, form the second layer of the Memento infrastructure. By virtue of this layering, Memento's semantic layer inherits the ability to support collaboration. Together, the two layers of the infrastructure enable collaboration over meaningful information artifacts, called *models*. The implications of this novel capability are described in Section 3.6.2.

Recall from Chapter 1 that assistants are applications that serve as adjuncts to the human mind. To be useful, an assistant must have an understanding of some domain of knowledge that it can apply on a user's behalf. The assistant must also communicate with its users in a meaningful way. Communication of meaning occurs through the semantic level of the user interface, described in Section 1.3. Such communication is centered around models derived from the assistant's domain. Each assistant's user interface may be tailored explicitly for its domain. However, all assistants, regardless of domain, have the requirement of communication and understanding of their models.

Models are not the only things shared between an assistant and its user. Both must share a common understanding of the domain from which the models arise. A domain can be characterized by a set of membership rules, called *semantic constraints*, over all possible models. A *domain* is the subset of all possible models that satisfy the semantic constraints. The semantic constraints of an assistant's domain must be represented—either programmatically or declaratively in the assistant. An assistant's domain may be a simplification of a real-world domain already familiar to a user. Alternatively, an assistant may serve as teacher and coach for users that may be unfamiliar with its domain.

If Memento is to support the needs of a wide variety of assistants, it must abstract the handling of an assistant's models and domain. Specifically, the Memento infrastructure must:

- represent models from a wide variety of domains in a uniform manner;
- have an effective understanding of a given assistant's domain and the semantic constraints associated with it;
- facilitate meaning-preserving manipulation of models from a particular domain;
- enable the evolution of domains over the lifetime of an assistant; and
- aid an assistant in the communication of its models through its user interface.

This chapter is divided into sections which address each of these four issues in turn. The relevant mechanisms of the Memento infrastructure are described in each section. Section 3.6 concludes the chapter with a discussion of the implications of these mechanisms and an assessment of how well the Memento infrastructure supports the semantic needs of assistant applications.

3.1 Model Representation

In order for Memento to support a variety of assistants, it must first represent models from a wide variety of domains in a uniform manner. This section details how models are represented in Memento. Models are *conceptual graphs*, an established knowledge representation scheme which is similar to the well–known, Entity–Relationship model. Conceptual graphs are composed of *concepts* and *relations*. Memento represents these components in terms of MObjects. The MObject data model was described in Section 2.1.

The reader is not assumed to be familiar with conceptual graphs. Relevant portions of conceptual graph theory will be described here. Additionally, the discussion will highlight the modifications and simplifications of the theory have been made for adaptation with Memento.

The following sections describe first how models are represented in terms of conceptual graphs and then how general conceptual graphs are represented by Memento.

3.1.1 Models are Conceptual Graphs

Memento relies on the established knowledge representation scheme of conceptual graphs for representing an assistant's models. This section will describe the rationale for using conceptual graphs, key features of the conceptual graph model, and how it compares with the Entity–Relationship model. Whereas this section concentrates on the abstract conceptual graph data model, Section 3.1.2 describes the more concrete aspects of conceptual graph usage in Memento.

Reasons for Using Conceptual Graphs

Memento's representation of models are based on John Sowa's *conceptual graphs* as described in his book *Conceptual Structures* [103]. There are several reasons for this choice. First, it is not the intent of this dissertation to invent a new knowledge representation. An existing, mature representation is strongly preferable to a new one. The study of conceptual structures has continued for more than a decade and has spawned a yearly conference on the subject [82, 90]. Next, during early experimental work on Memento, a representation scheme was tried that allowed the sharing of information *nodes* in the context of one or more other nodes. Nodes were linked by contextual relationships called *elements*. After discovering conceptual graphs, all earlier work was recast into the conceptual graph representation by a simple nomenclature change. Thus, conceptual graphs proved an excellent match for the work already in progress. Finally, conceptual graphs have a strong basis in cognitive psychology. Sowa gives compelling evidence that something like conceptual graphs are built and remembered during cognitive processing. Something akin to these graphs effectively forms the basis of knowledge and information processing in the mind. By using a computer representation for models resembling that of the mind, the chance of information loss is minimized in the translation between user and computer.

Key Features of Conceptual Graphs

A conceptual graph is composed of *concepts* and connected by *relations*. Type lattices over concepts and relations capture the possible "is-a" relations between concept types and relation types. Concepts, relations, and their type systems are now discussed in turn. Section 3.1.2 describes how both concepts and relations have attributes that will allow Memento's models, which are conceptual graphs, to carry their information content.

In conceptual graph theory, *concepts* are constructed by the mind as a representation for percepts gathered by the senses. "Run," an indefinite action; "Fido," a specific dog; "butter," a continuous substance; and "cold," a sensory perception are all examples of concrete concepts. Additionally, concepts can arise from abstract constructions that do not have analogies in the real world. "Orangeness," "freedom," and "infinity" are all examples of abstract concepts. All nouns are concepts, as are many verbs. In a conceptual graph diagram, concepts are represented as rectangles that are usually labeled with the concept's type and an optional identifier, if it is definite.

This dissertation is not concerned with whether or not a concept represents a real–world entity or how the mind would construct or manipulate such a representation. The concern is with concepts as they are represented in the computer and how graphs containing concepts can be manipulated in meaningful ways. Indeed, the concepts represented in Memento may have a visual representation in the user interface of an assistant. The visual representation gives rise to percepts in the mind of an assistant's user that are translated into concepts there.

A *relation* represents a direct association between two or more concepts. A relation appears as a circle in a conceptual graph drawing. It has directed edges to or from its associated concepts. All edges must connect to some concept, but multiple edges may connect to the same concept; the edges are tightly bound to its relation. An n–ary relation has edges labeled 0 through $n-1$, usually inside the circle. The 0th edge is always an in–edge. All others are out–edges. Most relations are binary and hence have one in–edge and one out–edge. In this situation, the numerical labels are omitted as they are redundant. As with concepts, each relation is labeled with a type and an optional identifier.

Each numbered position has a fixed meaning that is intimately tied to the type of the relation. The numbering simply defines the role of the associated concepts with respect to the relation. For example, a "Part" relation might represent a relationship between a whole and one of its components. The "Part" type would dictate the convention that the relation's in–edge attaches to the association's whole and the out–edge attaches to the association's component. What is important is not the particulars of the assignment, but that the assignment is everywhere consistent.

Figure 3.1 is a conceptual graph for a block world arch comprised of three blocks. All relations in the graph are binary, hence the edge labels are omitted.

The final key feature of conceptual graphs is a type system over concepts and relations. The conceptual graph type system over concepts is a type lattice that seeks to describe the possible "is–a" relations between concept types. For example, "spoon," "fork," and "knife" are considered subtypes of "silverware." "Knife" is a

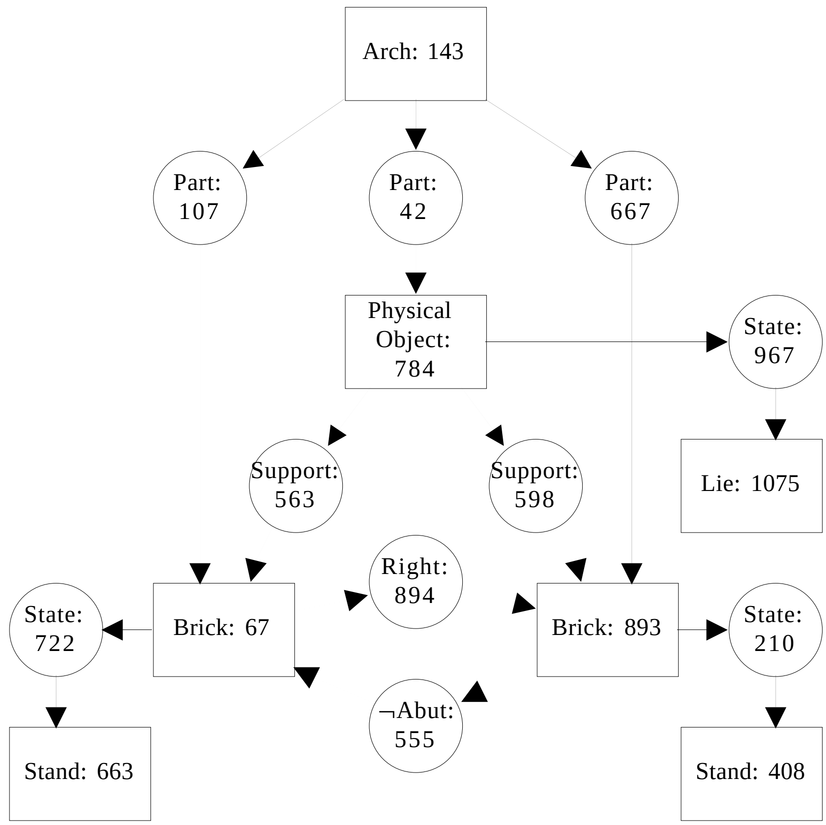

Figure 3.1 A conceptual graph for an arch adapted from [103], p. 71

subtype of "weapon," as well. A similar, but less rich type lattice over relations describes the possible "is–a" relationships over relation types.

In conceptual graph theory, the type lattices over concepts and relations are derived from established linguistic categorizations. They are essentially fixed and tacitly agreed upon. As such, the type lattices are not usually represented as part of conceptual graph diagrams. In conceptual graph theory, the type lattices primarily serve the function of deciding when two concepts or two relations, in fact, represent the exact same thing. One of the simplifications of conceptual graph theory that has been made for Memento is that the type systems over concepts and relations are hierarchies and not lattices. Section 3.1.2 describes this simplification and others in more detail.

Comparison with the Entity–Relationship Model

The Entity–Relationship model, first proposed by Peter Chen in 1976 [22], is a well–known data representation scheme which closely resembles conceptual graphs. Concepts correspond closely with entities, relations with relationships, and relation positions with roles. The Entity–Relationship model allows for attributes on both entities and relationships; whereas attributes attached to concepts and relations are not explicitly a part of conceptual graphs. By contrast, the Entity–Relationship model does not admit to a system of types over entities or relationships. Such a type system has been subsequently added to the model, however [106].

Entity–Relationship diagrams from the model are graphs that characterize all possible entities and relationships in a specific data set. The diagrams include information about whether (binary) relationships are 1 to 1, 1 to n, or n to m. The corresponding idea from conceptual graphs, *schema*, seeks to describe representative or characteristic conceptual graphs. Instead of limiting the possibilities, schemata describe what minimal features a conceptual graph must have to match the desired higher–level concept.

Memento borrows some ideas from the Entity–Relationship model. Specifically, attributes are added to concepts and relations and cardinality ranges are admitted in Memento's *metamodels*, which serve a similar purpose as schemata over conceptual graphs. Memento's philosophy of use is much more in keeping with that of conceptual graphs, however, so conceptual graph ideas and terminology are emphasized in this work. How Memento diverges from the traditional conceptual graph model is described in the next section. *Metamodels* are described in Section 3.2.

3.1.2 Conceptual Graphs in Memento

Memento represents both concepts and relations from conceptual graphs as MObjects with the `Concept` and `Relation` MTypes, respectively. Both MTypes are immediate subtypes of the `MObject` MType. The `Concept` MType requires no additional attributes. The `Relation` MType requires an `inConcept` attribute which contains the ReferenceDescriptor of the Concept that connects to it in the zero position. Additionally, the `Relation` MType requires an `outConceptList` attribute which is a list of ReferenceDescriptors for each of the positive numbered positions associated with the Relation. Note that back–pointers from concepts to relations for enumerating the sets of Relations associated with it are not explicitly represented. Instead, these pointers are calculated for the context in which the Concept is taking part. The set of such pointers that an assistant "sees" is always a subset of the actual relations adjoining the concept, effectively making a context–sensitive view.

Any subtypes of `Concept` or `Relation` that are needed by an assistant can be created by the same mechanisms as any other MType that are required by an assistant. The creation of an assistant's MType system was described in Section 2.4.3.

Adaptations of the Conceptual Graph Model for Memento

Because Memento's purpose for using conceptual graphs differs somewhat from their original intent, several simplifications and modifications have been made in Memento's use of conceptual graphs. First, indefinite concepts and relations are not allowed in Memento. All Memento conceptual graphs are composed of definite concepts and relations. Additionally, Memento does not allow shorthands for aggregations. Next, Memento's MType system is a hierarchy so a complete conceptual graph type lattice cannot be expressed in Memento. Finally, a specific concept in a conceptual graph is often tagged as the "root" of that graph in Memento. Each of these issues and the reasons behind them are discussed, in turn.

First, all concepts and relations in Memento are considered to be definite. Much of the machinery in the established theory of conceptual structures deals with linguistic analysis of written text. Usually, a conceptual graph is constructed for each sentence in a passage. The resulting graphs are then combined in a stepwise procedure. Often, a conceptual graph structure built in this process contains indefinite concepts. Such concepts act as placeholders for information that is usually gathered later. Consider the following passage:

> The cat stretched then sprawled out on top of the television. Despite the noise, Muffin was quite comfortable sleeping there.

A conceptual graph for just the first sentence would have to contain a concept node for an indefinite "the cat." A conceptual graph for the second sentence would contain a definite concept with the cat's name without the knowledge of what it really is. When the two conceptual graphs are correctly combined, these two concepts are unified into a single concept having type "Cat" and identity "Muffin."

A different set of assumptions is made with assistants. Because users of Memento–based assistants effectively manipulate a repository of existing models (conceptual graphs), the problem of unifying conceptual graphs into a whole is rare. The whole already exists and the concern is adding to it or modifying it in a meaningful way. Memento assumes that individual concepts and relations will never have more than a single representation in the repository. This is called the *uniqueness property.* From this property one can assume that every model is fully definite. If a placeholder is needed, a definite concept or relation is placed in the graph with as much information as is available. More information can be added later, as it becomes known.

Next, aggregation is handled in a different way than conceptual graph theory normally allows. Conceptual graphs are allowed to use a *set* of concepts in place of a single concept in a conceptual graph. The set, which may be only partially specified, makes it easy to build and manipulate conceptual graphs for sentences containing plurals and specified groups. The two most common cases of using sets in conceptual graphs are *collective sets* and *distributive sets.* In a *collective set*, all elements of the set participate in its relations together. For example, "We bought that couch in 1984." In the case of a *distributive set*, all components of the set participate in a relation, but they do so separately. For example, "Bill and Al work for the United States government."

Memento does not allow a set to take part in a conceptual graph in this way. Explicit specification is instead required. In the case of a collective set, a single concept representing the group would instead be created to play the part of the set. This group concept node would then be explicitly connected to concept nodes for each known member of the set via "contains" or "has–a" relations. Figure 3.2 shows how a conceptual graph for a collective set might be transformed for use by Memento. In the case of a distributive set, the set is replaced by concepts representing each element in the set and replicates the relation, and sometimes the attached concept, for each. Figure 3.3 shows how a distributive set might be transformed for use by Memento.

The next difference between the general conceptual graph theory and its application in Memento concerns the type system. Conceptual graph theory allows for types over concept and relations that are lattices. Because of the conscious decision to limit Memento's MType system to a hierarchy, only a hierarchy of types for concepts or relations can be represented. This is considered a limitation of Memento's first layer.

The final minor difference between conceptual graph theory and Memento is that in Memento, a single concept node is often designated as the "root" of the conceptual graph. The designation does not play a part in the semantics of the graph, but allows a convenient handle for navigation and specification. The root of a conceptual graph is indicated by an arrow pointing to the concept's rectangle. Later, a set of

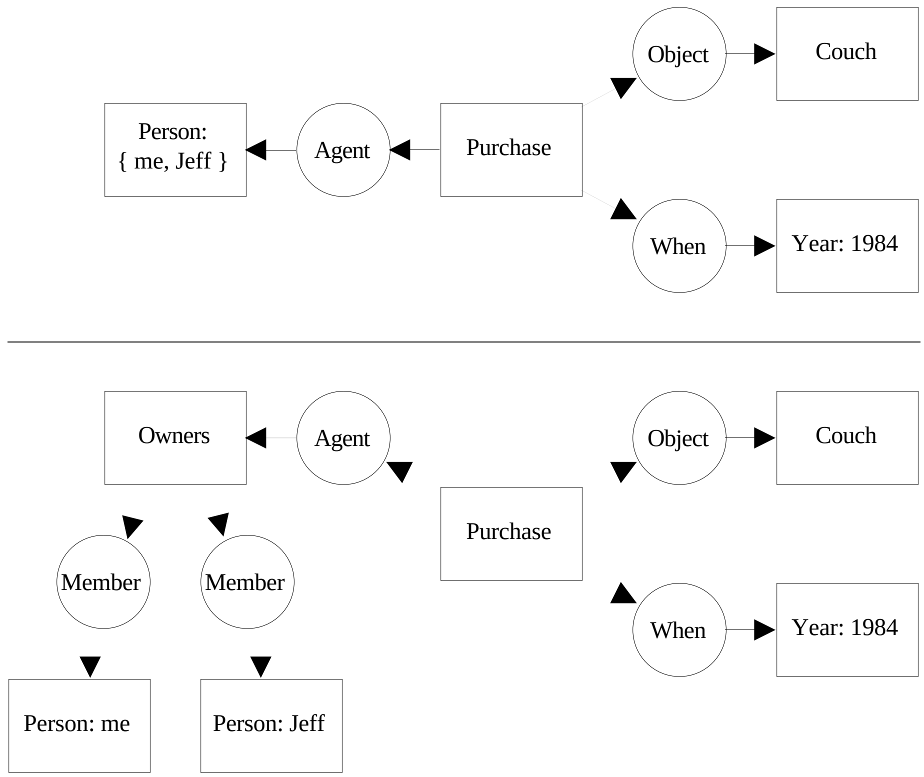

Figure 3.2 An example of how a collective set might be translated for use with Memento

graphs recognized by a given conceptual graph pattern can be specified by simply enumerating the set of roots of such graphs with the details of how those graphs match the pattern abstracted away. Note that Memento's conceptual graph patterns, called *metamodels*, will have a designated root, as well.

3.2 Metamodels: An Explicit Domain Representation

In order for Memento to be a useful infrastructure for assistants, it must have an effective understanding of a given assistant's domain and the semantic constraints associated with it. This section will describe an explicit domain representation, called a *metamodel*. A metamodel expresses a set of conceptual graph connectivity and type constraints and acts as a pattern for specific conceptual graphs. An assistant specifies its domain by enumerating a collection of metamodels that embody the assistant's

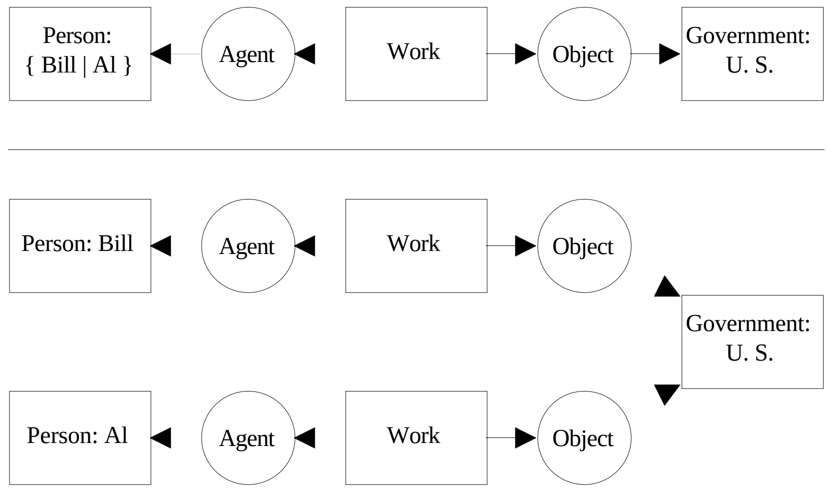

Figure 3.3 An example of how a distributive set might be translated for use with Memento

semantic constraints. Conceptual graphs recognized by an assistant's metamodels are the assistant's *models*. Metamodels are represented by MObjects. The `MObject` subclasses in Memento that comprise a metamodel embody an incremental model recognition algorithm.

3.2.1 Metamodels are Patterns over Conceptual Graphs

In conceptual graph theory, a certain type of conceptual graph, called a *schema*, can act as a pattern or frame for a higher–level linguistic or mental construct. It is through such schemata that "domain–specific knowledge about the typical constellations of entities, attributes, and events in the real world"[17] are expressed. A *schema* is an abstraction for a single concept that places the concept in the context of a conceptual graph. The body of the schema is a conceptual graph that acts as a pattern of interconnections to a concept that must be matched for it to be considered part of the schema's *type*. Depending on the specific use of conceptual graph theory, a given conceptual graph may not be required to exactly match the schema, but instead satisfy some measure of closeness. In other applications, a high–level concept may be

[17] [103], p. 128.

required to match a set of schemata called a *schematic cluster*. For the sake of this dissertation, exact matches of single schemata will be required.

Schemata effectively extend the type system over concepts for conceptual graphs. The addition of a schema effectively creates a subtype of the type the schema requires for its abstracted root concept. This type system created by interrelated schemata can be thought of as a *contextual type system* that places requirements on the context in which a concept of that type must appear. Types constructed by schemata can be used as the type specification of a concept in a conceptual graph or another conceptual graph schema. Schema definitions can thus embody a great deal of complexity by reference to other schemata.

Metamodel Definition

Memento's schemata are called *metamodels*. "Metamodels" is used in place of "schemata" for two reasons. First, the name emphasizes that the purpose of metamodels is to recognize conceptual graphs that are models from an assistant's domain. The other reason is that a metamodel differs somewhat in its definition from a conceptual graph schema. One simplification and one generalization to conceptual graph schemata have been made. Schemata have been simplified in that all concepts and relations in a metamodel are indefinite. By contrast, conceptual graph schemata allow definite concepts and relations. Whereas an indefinite concept would match any concept of its type, a definite concept would be required to additionally match the concept's identity. Recall the additional restriction that all concepts and relations in (non-pattern) conceptual graphs are definite.

Memento's metamodels generalize conceptual graph schemata in that they allow the expression of aggregation (branching) in the recognized conceptual graphs. Metamodels specify aggregation through the addition of cardinality ranges that restrict the branching that is allowed at various points in a pattern graph. With the allowance of aggregation, concepts and relations from a metamodel may ultimately match a set of concepts or relations from a single recognized graph. In conceptual graph schemata, where aggregation is not allowed, a concept or relation in a schema correlates with exactly one concept or relation in a recognized conceptual graph. For this reason, concepts and relations appearing in a metamodel are called *metaconcepts* and *metarelations*, respectively.

A *metamodel* is a rooted conceptual graph pattern that is composed of *metaconcepts* that are linked by *metarelations*. The metamodel graph is usually specified in the form of a conceptual graph–like diagram and enclosed in a box that is labeled with the name of the metamodel. A single metaconcept from the graph serves as the root of the pattern. As with rooted conceptual graphs, a metamodel's root is indicated by an arrow.

A *metaconcept* can be thought of as a contextual pattern for a set of concepts. In a metamodel graph, a metaconcept appears as a raised rectangle that is labeled with a type. The type may be any normal concept type or a reference to another metamodel (a contextual type). In either case, the metaconcept *matches* all concepts having that type.

As with conceptual graph schemata, the type the metamodel defines is a contextual subtype of the type specified in its root metaconcept. For subtyping to make sense, the type specified in the root metaconcept must be *grounded* with respect to the metamodel. That is, the type specified there must be either a normal concept type or a contextual type (metamodel) that is simpler than the one being defined. For example, a root metaconcept may not refer to its containing metamodel or any other metamodel that is a contextual supertype of it. The containing metamodel is allowed to appear as the type of any non–root metaconcept, however. Such constructs are useful for specifying structural recursion in conceptual graphs.

A *metarelation* can be thought of as a contextual pattern for a set of relations. Metarelations appear as raised circles in metamodel graphs. They are labeled with a relation type. (Memento has no contextual type system analog for relations.) Metarelations are connected to metaconcepts by the same rules that connect relations to concepts. Connecting edges are also directed and labeled in the same way. A metarelation *matches* the set of all existing relations that:

- connect to concepts that are matches of the adjoining metaconcepts in the corresponding edge positions, and
- are of a subtype of the metarelation's labeled type.

Exactly one of a metarelation's edges must have a cardinality range adornment. A metarelation's cardinality range adornment indicates where branching may be introduced in the conceptual graphs recognized by a metamodel. The adornment at an edge indicates the upper and lower bound on the number of relations of that type that may appear attached to a particular concept. The cardinality range is specified by a lower and upper bound, for example, "7–20." The usual shorthands are allowed:

a single number when the lower and upper bounds are the same, an asterisk (*) for an unrestricted range, and a plus sign (+) for a range of at least one. The adornment may be on the metarelation's in–edge or any of its out–edges. The exact definition of this adornment will be stated in terms of metaconcept and metarelation *satisfaction* sets described below. But before satisfaction can be properly defined, however, another idea must first be introduced.

Every metaconcept in a metamodel must be able to be placed in a total ordering with respect to adorned edges in the metamodel graph. The root metaconcept must be first in the ordering. For every metarelation in the metamodel, the metaconcept attached to an adorned edge must be earlier in this ordering than any of the metaconcepts attached to unadorned edges of the metarelation. This mathematical relation between metaconcepts is distinct from and unrelated to the edge directions connecting the interposed metarelations. Any metamodel graph not having such a total ordering is considered invalid. The total ordering restriction is necessary to guarantee termination of the model recognition algorithm, described later in this section.

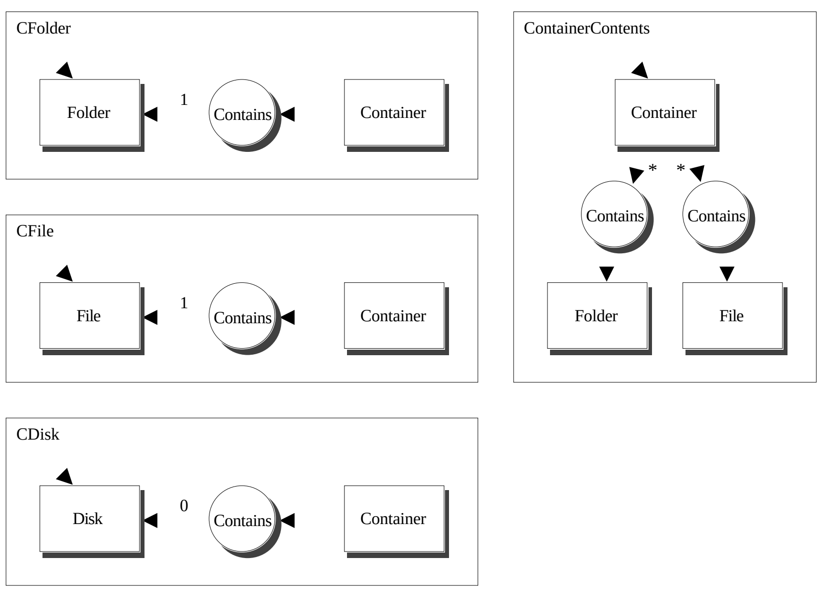

Figure 3.4 Metamodels describing file system containment relationships

Example metamodels based on the familiar Macintosh file system and "desktop metaphor" are now presented. Within the file system, a disk contains folders and files. Folders can also contain folders and files. It makes sense, then, to place the

"Folder" and "Disk" types under the common type "Container." "Container" and "File" would be concept subtypes. Additionally, a relation subtype, "Contains," is needed for file system containment. Figure 3.4 shows four metamodels describing the following rules for containment relationships in the hierarchical file system:

- A contained folder, or "CFolder," has exactly one container.
- A contained file, or "CFile," has exactly one container.
- A "CDisk" is a top–level container.
- A "ContainerContents" is a container that has any number of folders or files.

The first three metamodels describe proper containment relationships. The last metamodel might be used to effectively query the contents of a container.

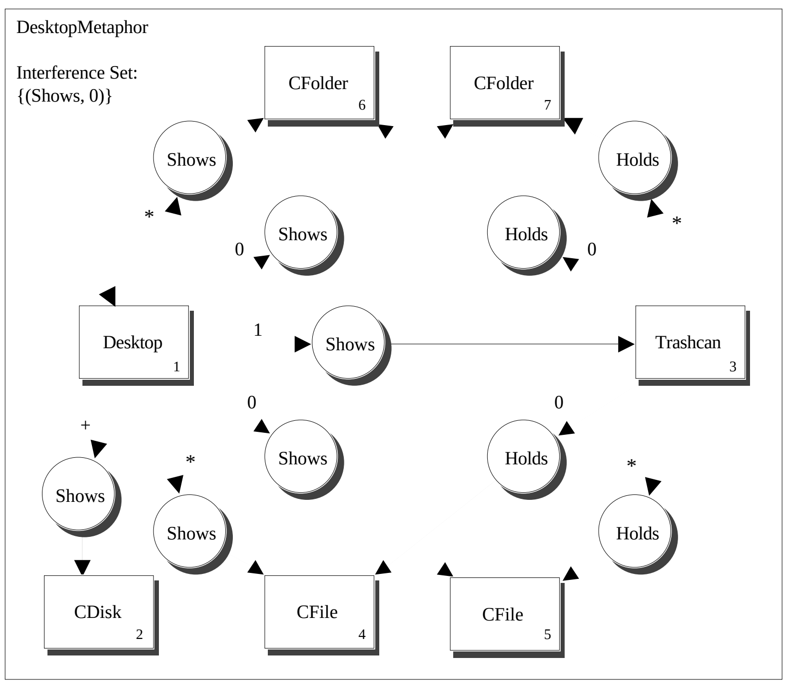

Figure 3.5 A metamodel showing the file system components appearing on a desktop

The Macintosh operating system allows file system components to appear on the "desktop" or main screen. Additionally, it introduces the notion of a trashcan, a holding place for files and folders to be deleted. Concept subtypes "Desktop" and "Trashcan" characterize the new entities. The relation subtype "Shows" characterizes the relationship between a desktop and something appearing on it. Similarly, the relation subtype "Holds" characterizes the relationship between a trashcan and a discarded object. In Figure 3.5, is a metamodel describing the relationships between the possible objects on the desktop. The following semantic constraints are captured:

- A desktop must show at least one disk.
- A desktop must show exactly one trashcan.
- A desktop can show any number of files and folders.
- A trashcan can hold any number of files or folders.
- No file or folder can be showing on the desktop and held by the trashcan at the same time.

The figure shows one possible total ordering of its metaconcepts by the numerals in the lower right hand corner of each metaconcept. The metamodel's interference set, described below, contains one pair.

The ideas of metaconcept and metarelation *satisfaction* are now defined. They are defined in terms of each other and in terms of the simpler metaconcept and metarelation *matching*. Whereas *matching* refers to the local context of a metaconcept or metarelation, *satisfaction* is a global property that is affected by the topology of the entire metamodel. Relations that *satisfy* a metarelation must:

- *match* the metarelation,
- attach to concepts that *match* the metaconcept for the adorned edge, and
- attach to concepts that *satisfy* the metaconcepts for all unadorned edges.

Concepts that *satisfy* a metaconcept must first *match* the metaconcept and then for each metarelation that attaches to the metaconcept through an adorned edge, the cardinality of the set of relations that attach to that concept and that satisfy the metarelation must be within the range specified by the cardinality range adornment on the edge. For metaconcepts that lack attached adorned edges, the satisfying set of concepts is the same as its matching set.

The set of concepts satisfying the metamodel's root metaconcept are, *by definition*, the roots of conceptual graphs that are recognized by the metamodel. The details of exactly how each graph satisfies the metamodel can be easily retrieved by starting from the graph's root concept and enumerating the concepts and relations based on the metamodel's satisfaction sets in forward adornment order. Because a concept that satisfies the root metaconcept of a metamodel describes an entire conceptual graph in the context of that metamodel, only a set of concepts need be exported from a metamodel as recognized instead of entire conceptual graphs. Recall that a metamodel's set of roots of recognized conceptual graphs effectively serves as the set of *matching* concepts for any metaconcept that refers to the metamodel as its (contextual) type.

Model Recognition Algorithm

An algorithm to determine the conceptual graphs (models) that adhere to a metamodel follows directly from the definitions for metaconcept and metarelation matching and satisfaction, given previously. Because metamodels are often defined in terms of other metamodels, one must first determine the transitive closure of all metamodels mentioned in the target metamodel. Additionally, for each metamodel, the total ordering over of metaconcepts over adornment edges should be known. The ordering can be determined once at the time of metamodel creation. The algorithm recognizes conceptual graphs (models) for all metamodels simultaneously. It proceeds in a bottom–up fashion and converges on the correct result over several iterations.

The algorithm starts with the metaconcepts that have simple (non–contextual) types. The matching sets for each of these metaconcepts is determined solely by the normal concept type. The matching sets for metaconcepts based on contextual types are initially set to empty. Once these sets are determined, the matching sets for all metarelations are determined based on the current values of the matching sets of the metaconcepts. The matching sets of metarelations only depend on the matching sets of their attached metaconcepts. They can be calculated in any order.

Next, for each metamodel the satisfaction sets for all metaconcepts and metarelations are calculated. This calculation relies on the fact that the satisfaction set of a metaconcept is only affected by metaconcepts later in the total ordering over adornment edges. Starting with the last metaconcept in the ordering, the satisfaction set of each metaconcept is calculated in reverse order. Because the last metaconcept in the total ordering cannot have any attached adornment edges, its satisfaction set is the same as its matching set. For all subsequent metaconcepts in the order, the set of attached adornment edges is determined, the satisfaction sets for their metarelations are calculated, and then the satisfaction set of the metaconcept is calculated.

Next, for each metamodel, the root metaconcept's satisfaction set is used to update the matching set of every metaconcept that mentions that metamodel's (contextual) type. This step must additionally determine whether this update actually changes the metaconcept's matching set. If there is no metaconcept in a metamodel whose matching set is changed, there is no need to recompute the matching and satisfaction sets for that metamodel. If there are updates, however, the metarelation matching sets must be recalculated and the satisfaction sets recomputed as before. The algorithm terminates when there are no longer any updates to any of the metamodels.

Some additional validity constraints on metamodels are required to guarantee that the the model recognition algorithm terminates. The difficulty stems from the fact that the satisfaction sets do not monotonically converge on their terminal values. For example, an addition to one metaconcept satisfaction set may trigger the violation of an adornment upper bound for a metarelation attached through an unadorned edge and subsequently cause the removal of a concept from another metaconcept satisfaction set. A pathological case can arise when one or more mutually recursive metamodels "interfere" with one–another—where the satisfaction of a conceptual graph of one metamodel will lead to a reduction in another's satisfaction set and vice versa.

To guarantee termination, the notion of a metamodel *interference set* is needed. A metamodel's *interference set* is a set of pairs, relation type and relation edge number, that the metamodel lists as possibly connected to the root concept of a recognized conceptual graph. This set must even include the relations that are explicitly disallowed in a metamodel by an empty adornment cardinality range (i.e. 0–0). The interference set of a metamodel is easily calculated from the set of metarelations attached to its root metaconcept. If the root metaconcept has a contextual type, the interference set from that type augments the interference set of the enclosing metamodel. Determining each metamodel's interference set is straightforward and guaranteed to terminate because of the contextual type system's groundedness property, mentioned earlier.

A metamodel's interference set is used to restrict the allowed metarelations that can attach to a metaconcept of the metamodel's type. Specifically, in any context where such a metaconcept appears, there may not be any attached metarelations that would duplicate any of the pairs appearing in the interference set. With the addition of this restriction, the model recognition algorithm is guaranteed to terminate.

The model recognition algorithm lends itself well to incrementality. An incremental version of this algorithm is part of the Memento infrastructure. This incremental algorithm has not produced a noticeable degradation of performance. The initial construction of a metamodel's matching and satisfaction sets can be somewhat expensive, however. Fortunately, this calculation need only be done when the metamodel is first constructed. The life cycle of a metamodel is discussed further in Section 3.4.

3.2.2 Metamodels Define a Domain

An assistant's domain is defined primarily by a group of metamodels. Recall that a domain is the set of models that are meaningful to an assistant. Through a mechanism described in Section 3.4, an assistant enumerates the metamodels that it requires. The metamodels capture an assistant's semantic constraints over model connectivity and component MTypes. Additional, ad–hoc semantic constraints must be hard–coded into subclasses of the `Model` class, described in Section 3.3. For now, the focus is primarily on the model connectivity constraints embodied in a metamodel.

Memento maintains a continuous and incremental recognition of all models associated with all metamodels that are known to it. An assistant need only select the models that it will display or manipulate from the recognized sets. The exact nature of how the metamodels are used or what they describe is determined by the assistant. Examples of Memento–based assistants and their associated metamodels are given in Chapter 4. Section 3.3 describes how models are manipulated in the context of their metamodels to preserve their meaning.

3.2.3 Metamodels are Constructed from MObjects

Metamodels, and their components, metaconcepts and metarelations, are represented by MObjects in Memento. Specifically, `Metamodel`, `Metaconcept`, and `Metarelation` MTypes are all immediate subtypes of the `MObject` MType. The `Metamodel` MType requires attributes for its name, its root metaconcept, the set of contained metaconcepts, the set of contained metarelations, and the set of recognized conceptual graph root concepts. The `Metaconcept` MType requires attributes for the containing metamodel, the associated type, the set of metarelations attached by in–edges, the set of metarelations attached by out–edges, the set of matching concepts, and the set of satisfying concepts. The `Metarelation` MType requires attributes for its containing metamodel, the associated type, the set of metaconcepts attached by in–edges, the set of metaconcepts attached by out–edges, the single adorned edge, the adornment cardinality range, the set of matching relations, and the set of satisfying relations.

Subclasses of the `MObject` class for these three MTypes embody the incremental model recognition algorithm. The algorithm makes extensive use of the MObject change notification mechanism of Memento's first layer. The mechanism, described in Section 2.3, allows, for example, a `Metaconcept` object to observe the MType system for changes to the set of MObjects associated with its reference type. Changes to this set are copied into the `Metaconcept`'s matching set. Similarly, a `Metarelation` object will observe its attached `Metaconcept` objects in order to incrementally calculate its matching and satisfying sets. All such notifications are done in response to *trial* change notifications of the observed MObjects. The net effect is that any change to any `Concept` or `Relation` automatically updates any metamodels that might be affected by the change.

Dynamic structural modifications of metamodels are allowed and are an important part of a domain's evolution. This subject is discussed further in Section 3.4. The `isValid()` method of the `MObject` class is overridden in the `Metamodel`, `Metaconcept`, and `Metarelation` classes so that structural modifications to any metamodel graph adhere to the restrictions of any conceptual graph and the additional restrictions on metamodels mentioned in this section. Recall that every MObject involved in a transaction must be valid at transaction commit time for the commit to be successful.

3.3 Preservation of Model Meaning

One of the most important functions that Memento provides an assistant is a guarantee that its semantic constraints are not violated during the manipulation of its models. Memento's meaning preservation mechanism is provided primarily by the `Model` class and an associated metamodel that characterizes the connectivity constraints over a set of models. The mechanism can be customized to admit ad–hoc semantic constraints through domain–specific subclasses of the `Model` class. Acting in concert with an associated instance of the `Metamodel` class, a `Model` subclass monitors transactions and blocks those that violate the assistant's semantic constraints. Memento's meaning preservation mechanism relies heavily upon its model recognition algorithm, which was described in Section 3.2.1. In addition to its meaning preservation function, the `Model` class has many other responsibilities as it is an assistant's primary interface to models recognized by a specific metamodel.

3.3.1 `Model` Subclass Instances Interface to a Model

An instance of Memento's `Model` class acts as an interface to a model that has been recognized by a particular metamodel. Objects of the class maintain a mapping between the concepts and relations of the model and their corresponding metaconcepts and metarelations in the metamodel. Assistants make use of this mapping, for example, to determine the relations attached to a given concept in the context of the metamodel. When an instance of the class is created, the class realizes all concepts and relations in the model for convenient access by the assistant. Realization of MObjects was discussed in Section 2.5.2. The `Model` class also generates high–level change notifications concerning how the model is modified by a transaction. These high–level notifications are discussed further in Section 3.5. Perhaps the most important responsibility of the `Model` class is to assist in the preservation of the model's meaning during an assistant's modification of it. How Memento preserves model meaning is described below.

Typically, a subclass of the `Model` class will be derived for interaction with models recognized by a particular metamodel. Such a subclass effectively embodies all necessary domain–dependent capabilities by extending the domain–independent capabilities of the `Model` base class. A `Model` subclass might provide convenience methods for accessing and modifying the model in a domain–specific way. Additionally, the subclass might provide domain–specific high–level change notifications. Finally, the subclass would be responsible for maintaining any ad–hoc semantic constraints that the assistant might impose on models from its domain.

3.3.2 Meaning Preservation During Model Manipulation

Memento's meaning preservation mechanism is embodied in the `Model` and `Metamodel` classes. These classes extend Memento's transaction mechanism described in Section 2.2. As mentioned above, the `Model` subclass can provide convenience methods for manipulating a model in the context of a particular domain. An assistant is not required to use those methods, however, for the manipulation of a model's components. It can use any of the `MObject` subclass modification methods directly. Additionally, another assistant can perform potentially damaging manipulations to an assistant's models if the two assistants share an MObject repository. Components of one model may appear in the models of a completely different metamodel, so even if every assistant "plays by the rules," trouble can arise. These two situations are now discussed in turn. Solutions to these situations prevent violations of an assistant's semantic constraints by seemingly drastic measures. Keep in mind that violations of this sort are rare in properly written and administered assistants.

Meaning Preservation in Assistant–Generated Transactions

For model manipulations initiated by an assistant, Memento's transaction mechanism is extended to require every transaction not to violate the assistant's semantic constraints. An assistant's semantic constraints fall into two components. Model connectivity and type constraints are embodied in a metamodel. Any remaining constraints are considered ad–hoc and must be hard–coded into a `Model` subclass. How each component participates in meaning preservation is now discussed.

Recall from Section 3.2.1 that an active metamodel continuously and incrementally recognizes its models by using the MObject change notification mechanism from Memento's first layer. The model recognition algorithm reacts to *trial* change notifications which occur during the course of a transaction. If the transaction adheres to the domain's type and connectivity constraints for a model, the metamodel will still list the model as recognized at transaction commit time. If the constraints are violated, the model will drop from the set of models recognized by the metamodel. At transaction commit time, the `Model` instance's `isValid()` method is called if any of its components is modified during the transaction. As part of its validity check, the method verifies that the associated models being used by an assistant (that is, models represented by `Model` instances) are still members of their recognized sets. If any are missing, the transaction is not allowed to commit and its effects are unwound.

An assistant's ad–hoc semantic constraints that cannot be captured in the context of a metamodel must be programmatically embodied in a `Model` subclass. There are two strategies for enforcing ad–hoc semantic constraints. First, a `Model` subclass instance can participate in the validation of a transaction by registering itself as desiring participation in the validation process of a particular transaction. A subclass of the `Model` class can override observation methods that allow it to know that some component of a model is being modified. In response, the subclass can request participation in the transaction's validation step. At transaction validation time, the subclass's `isValid()` method will be called and the ad–hoc semantic constraints over the model can then be checked. By returning a negative result, the subclass can effectively disallow transactions which would violate the assistant's ad–hoc semantic constraints. As an alternate strategy, a `Model` subclass can watch the trial modifications of components of its model and compensate for violating changes by amending the transaction in progress. Recall from Section 2.3.4 that any software component reacting to *trial* MObject change notifications can produce additional MObject change notifications that are part of the same transaction.

Meaning Preservation in Repository–Generated Transactions

A different strategy is required for transactions that originate from a repository. Recall from Section 2.5.4 that repository–generated transactions are the means by which an assistant becomes aware of MObject modifications made by other assistants. Invalidating these transactions does not have the effect of unwinding the modifications in the repository—such modifications are already permanent. An invalidation would only prevent the assistant from synchronizing with the repository and would force the assistant's termination. Instead, the assistant must react to a notification sent out by the affected `Model` instance indicating the model is no longer part of the assistant's domain. The assistant must gracefully react to such a notification by ceasing all use of the model. Any user interface component that is communicating the model to the assistant's user must be gracefully disabled so that no further modifications can be made. Immediately after the notification is given, the `Model` instance associated with the affected model is destroyed.

If an externally generated transaction does not violate an assistant's type and connectivity constraints but does violate an assistant's ad–hoc constraints, there are two strategies for preserving meaning. As with a connectivity violation, a `Model` subclass can simply indicate that the model no longer is in the domain of the assistant by emitting the same notification as before. Alternatively, a `Model` subclass can amend the transaction to correct the violation as it might do with an assistant–generated modification. In this case, the net change is propagated back to the repository for storage and notification of other assistants.

3.4 Metamodel Construction and Evolution

Because assistant's and their associated domains evolve, Memento must assist in the support of domain evolution. Memento does this through the explicit domain representation of metamodels. Metamodels are built from MObjects and are data. Like Memento's MType system, a metamodel that describes a set of models is present wherever the models themselves may reside. For example, a repository will contain all the metamodels for the models that are stored therein. As with Memento's MType system, there is a process of metamodel unification that must occur when an assistant connects to a repository in order to ensure that meaning is preserved across the connection.

There is a deep parallel between the data model and supporting classes of Memento's first level and those of its second level. The following facts are true about both layers of Memento. The details about these facts are summarized in Table 3.1.

- Information is carried by the main data object for the level which is ultimately represented in terms of MObject(s).
- The main data object is accessed through a Memento class of the same name.
- Subclasses of this class provide convenience access and modification methods and can provide customized change notifications.
- The level's meta–information constrains the main data object.
- The meta–information's constraints are respected during transactions.
- The meta–information is dynamic and represented ultimately in terms of MObject(s).
- Meta–information "lives" along with the data it describes.
- Meta–information evolves over time due to changes in the assistant's semantic constraints.
- An assistant characterizes its semantic constraints primarily in terms of declarative objects that are created by an assistant before run–time via C++'s static constructors.
- The declarative objects both construct the meta–information and prevent it from evolving beyond an assistant's semantic constraints during the lifetime of the assistant.
- An assistant's semantic constraints that cannot be captured in the declarative representation can be added programmatically to the subclasses of the main data object.

Section 2.4 described how an assistant's MType system is initially constructed and evolves over time. This section describes how an assistant's metamodels are initially constructed and evolve over time. While the details of these two life cycle descriptions differ, they are very similar in spirit. An assistant characterizes its model connectivity semantic constraints declaratively in instances of the `ModelClass`. When an assistant begins execution, metamodels are constructed from the available `ModelClass` instances. When an assistant connects to a repository, it must unify its metamodels with corresponding metamodels in the repository. The unification process might modify an assistant's metamodels, effectively altering the domain of the assistant. The `ModelClass` instances observe the metamodels over the lifetime of the assistant and ensure that the assistant's semantic constraints are not violated by a change to its metamodels.

3.4.1 `ModelClass` Instances Capture an Assistant's Model Connectivity Semantic Constraints

Instances of the `ModelClass` class are created by an assistant to characterize the features that its corresponding metamodel must have in order to be manipulated by the assistant. `ModelClass` objects are constructed in an assistant before run–time via C++'s static constructors. In essence, a `ModelClass` object describes the most general metamodel that can be meaningfully manipulated by the assistant. By contrast, the metamodel describes the connectivity semantic constraints that should be preserved for *a particular set of models.* While these two are usually the same, it is possible that the repository may actually have a more constrained data set than the assistant might normally generate. In all cases, the assistant must adhere to the constraints of its metamodels. This may mean that some operations must be dynamically disabled by an assistant's user interface. Section 3.5.2 will discuss how the domain of an assistant is communicated through its user interface.

Feature	Memento Level I	Memento Level II
Main data object	MObject	model
Memento class	`MObject`	`Model`
Meta–information	MType	Metamodel
Memento class	`MType`	`Metamodel`
Declarative constraint	`MObjectClass`	`ModelClass`

Table 3.1 A comparison of key features of Memento's two layers

3.4.2 Metamodels are Constructed from `ModelClass` Instances

The single `Memento` class instance of an assistant orchestrates the construction and eventual destruction of the assistant's metamodels. When an assistant begins execution, the `ModelClass` objects are gathered and checked for completeness and consistency. The metamodels are then constructed during a single transaction. At this time, the `ModelClass` instances register themselves as observers of their associated metamodels. They continue to monitor dynamic changes to the metamodels to ensure that a dynamic change to a metamodel does not prohibit it from being manipulated by the assistant. Immediately after construction, the incremental model recognition algorithm embodied in the `Metamodel` and component classes is activated. The recognition mechanism remains activated throughout the lifetime of the assistant. During the termination of an assistant, each of these steps is undone in reverse order.

3.4.3 Meaning Preservation During Metamodel Unification

Metamodels are intentionally dynamic. They are subject to modification just as any other construct comprised of MObjects. The most common source of metamodel modification is the process of unification with a repository. As with MTypes, each Memento repository contains a set of metamodels that describe the models therein. Unification ensures that an assistant's manipulation of the models from a repository preserves the intended meaning of the models. Metamodel unification occurs immediately after a successful MType unification. The process of repository MType unification was discussed in Section 2.4.3.

Successful metamodel unification requires the eventual equality between corresponding metamodels in the assistant and the repository. Unification proceeds in a pairwise comparison of metamodels by name. If a repository lacks a metamodel that exists in an assistant, a copy of that metamodel is created in the repository. Any extraneous metamodels in the repository are either ignored or copied into the assistant depending on whether they are mentioned in a metamodel that corresponds to one in the assistant. Unification of two metamodels is possible only if the two metamodels are the same or if one is a specialization of the other. If one is not a specialization of the other, the unification process fails and the connection to the repository is aborted.

Metamodel specialization, which is described in detail below, adds constraints to a metamodel in an orderly way. Be aware that it is not necessarily the case that adding constraints results in a smaller recognized model set. Specializing a metamodel, therefore, does not result in a contextual subtype of the original metamodel.

There are two cases: either the repository's metamodel is a specialization of the assistant's, or vice versa. Because it is assumed that domains become more specialized over time, an attempt is made to bring the more general metamodel into agreement with the more specialized one. In the case that the repository's metamodel is more specialized, the assistant's metamodel is simply modified to match the repository's metamodel. No special checking need be performed. Because of the nature of metamodel specialization, this process is guaranteed not to violate the assistant's semantic constraints embodied in its `ModelClass` objects. Recall that a `ModelClass` object observes all modifications to its corresponding metamodel and will invalidate any transaction that would violate its embodied connectivity constraints.

The case of specializing an existing repository metamodel is somewhat trickier. Recall that a repository's metamodel describes the constraints over a subset of the models stored there. Any change to a repository's metamodel could potentially remove some stored models from the set recognized by the metamodel. The resulting undesirable effect would be that some of the repository's models would be rendered meaningless. A restriction is therefore added that the set of models recognized by a repository metamodel during its specialization must not change. More specifically, the components of the models may change, but the set of root concepts of recognized models must not change. For metamodel specializations that admit the original set of models, nothing special need be done. For more complicated specializations, the offending models would need to be repaired by the same unification transaction. This latter case is potentially a very expensive operation which is best performed "off–line" by a specially constructed agent to assist with the evolution of a repository's models. In the situation were an assistant's domain evolves in a way that cannot be described as a specialization, another such agent would need to repair a repository's models and metamodel(s). Because metamodel specialization captures the most common evolutionary steps of a domain, such specialized agents are rarely required.

Once the metamodels from a repository and an assistant have been unified, the assistant can take advantage of the fact that the matching and satisfied sets of the metamodel's components are up–to–date with respect to the repository's concepts and relations. Each `Repository` object keeps its repository `Metamodel`, `Metaconcept`, and `Metarelation` MObjects synchronized with those of the assistant. The net effect is that the model recognition algorithm becomes distributed across all Memento executables sharing the metamodel. This dissertation does not address the thorny issues raised by an assistant connecting to more than one repository.

Metamodel Specialization

A metamodel, M', is a *specialization* of another, M, if there is a sequence of operations, enumerated below, that converts M into M' and results in a valid metamodel. Specialization operations have two important properties. First, they characterize the most likely evolutionary paths of metamodels. This implies that as an assistant evolves over time, its metamodels are most likely to change by a series of specializations. The second property is that specialization preserves most of the features of the original metamodel. Because of this property, Memento's `Model` class can continue to function in a meaningful way on a specialization of a metamodel for which it was originally intended. In effect, the second property is what allows an outdated assistant to still perform useful operations on models from a more evolved domain.

Be aware that a specialization of a metamodel is not necessarily a contextual subtype of the original. This means, in effect, that a specialization of a metamodel may recognize some models that are not recognized by the original.

Metamodel specialization operations include:

1. Replace a metaconcept's concept type with one of its subtypes. This requires more attributes on a corresponding concept in a recognized model.

2. Replace a metaconcept's type with a contextual subtype. This requires more connections to a corresponding concept.

3. Replace a metarelation's type with one of its subtypes. This requires more attributes on a corresponding relation.

4. Restrict the cardinality range of an existing adornment edge. This further constrains the branching at a concept.

5. Add an unattached metaconcept of any type. This is necessary as part of extending the metamodel graph. The metaconcept must eventually be connected into the metamodel for it to be valid.

6. Add a metarelation between (or amongst, in the case of a tertiary or higher relation) metaconcepts. This places additional constraints on model connectivity.

The new metarelation's adornment edge may not violate the total ordering restriction required by the model recognition algorithm. It may, however, change the actual ordering of metaconcepts within the total order.

Note that all metamodel specialization operations preserve the connectivity of the original metamodel.

3.5 Communication of Models and Domains through an Assistant's User Interface

The final capability that Memento must provide is to aid in the communication of models and domains through an assistant's user interface. These two issues are discussed in turn in the next two sections.

3.5.1 Communication of Models

Section 3.3 described how a subclass of Memento's `Model` class acts as an interface to a model recognized by a metamodel. A `Model` subclass usually provides domain–specific access and modification methods. These methods allow a user interface component to access a model's information for display or to perform modifications on a displayed model. The remaining capability, for a software component to react to dynamic model modifications, is also part of the `Model` class and its subclasses.

Recall from Section 2.3 that software components can register themselves as observers on MObjects and receive detailed notifications of changes to them. As with the MObject change notification mechanism, Memento's `Model` class dispatches change notifications that describe how a model's connectivity changes during the course of a transaction. These so called *high–level change notifications* occur in addition to the low– and intermediate–level change notifications dispatched by Memento's first layer. These notifications include:

- additions to, and removals from, the sets of concepts and relations associated with corresponding components in the metamodel;
- a change in a concept's (or relation's) associated metaconcept (or metarelation) in the case that the concept (or relation) remains part of the model but its role in the metamodel has changed; and
- notification that the model is no longer recognized by the metamodel.

Subclasses of the `Model` class can generate their own domain–specific change notifications in response to specific changes in the model's connectivity and dispatch them by the same mechanism.

3.5.2 Communication of a Domain's Semantic Constraints

An assistant has the additional job of communicating its domain via the user interface. An assistant's domain is characterized by its metamodels which embody connectivity constraints over the models they recognize. Metamodels are dynamic and may change during the execution of an assistant. Recall from Section 3.4 that an assistant's metamodels may be specialized during the process of unification with a repository. If this occurs, an assistant's metamodels describe a different (but related) set of models than the set that would normally be generated and manipulated by the assistant. For this reason, an assistant may not be able to perform all of its normal operations on a specialized model set. Communication of an assistant's domain is performed by features in the user interface that effectively describe what operations are possible. Often, domain communication implies, simply, the deactivation of user interface components, such as menu items, that are inappropriate for the specialized models.

Consider, for example, the operation of creating a new model that would be recognized by an assistant's metamodel. Some specialization operations applied to a metamodel during unification may render the assistant unable to create a recognized model. Some metaconcept or metarelation type specializations, for example, may require attributes on concepts or relations that cannot be generated by the assistant. The `Model` class provides a method to answer the question of whether it is even possible for the assistant to create a new, recognized model. Subclasses of the `Model` class can override this method and provide other methods that communicate domain–specific capabilities. A user interface component can call these methods to determine whether various operations can possibly succeed. This mechanism allows an assistant's capabilities to be gracefully disabled in situations where they can never succeed.

3.6 Implications of Memento's Second Layer

Memento's second layer provides the mechanisms to represent the models from a variety of domains, have an effective understanding of the models, preserve the meaning of models during their manipulation, facilitate the evolution of domains, and aid in the communication of models and domains through the assistant's user interface. Together, these mechanisms address the semantic needs of assistants. Several issues concerning these mechanisms deserve further comment. The issues of Memento's explicit domain representation are first discussed. Then, the issues of how Memento supports collaboration over models, the evolution of information spaces, and the construction of assistant applications are discussed. The chapter closes with a brief discussion of Memento's implementation status.

3.6.1 Implications of an Explicit Domain Representation

Memento uses an explicit representation of an assistant's domain in order to preserve the meaning of the assistant's models. The domain representation consists of a set of metamodels and an associated MType system—all of which are ultimately represented in terms of MObjects. The same mechanisms that manipulate, transmit, and store an assistant's models can also be used for the components of the domain representation. An explicit domain representation allows for Memento to preserve the semantic integrity of an assistant's models while, at the same time, facilitating the evolution of the domain. The issues of meaning preservation and evolution are now discussed in turn. The section concludes with some interesting potential uses for the domain representation that have yet to be fully explored.

Memento preserves the semantic integrity of its models by enforcing the constraints embodied in its domain representation. The transaction mechanism from Memento's first level is extended in Memento's second level to assure that a domain's constraints are not violated by the assistant. Through the process of metamodel unification, a domain representation stored with a set of metamodels is checked for consistency with the domain representation of an assistant. Such checking ensures that the meaning of the models in a repository is preserved during an assistant's manipulation of them. Domain representations are an important part of any Memento collaboration.

As with most applications, assistants evolve over time. Memento supports the evolution of assistants through its explicit domain representation. An assistant's type and model connectivity semantic constraints are expressed declaratively and are readily modified. From these constraints, an assistant's domain representation is created at the beginning of an assistant's execution. An assistant's domain representation may be altered during a connection to a repository through the process of metamodel unification. Through alteration of a domain representation, an assistant is able to dynamically deactivate capabilities that are inappropriate for a more highly constrained set of metamodels. Such a situation might arise when an older assistant tries to manipulate models that have been created by a newer, more evolved one.

Other purposes for the domain representation have been envisioned, but have not yet been realized. First, because a stored domain representation characterizes the models in a repository, it could be used for MObject garbage collection. Stored MObjects could be safely purged if they were not part of conceptual graphs recognized by metamodels in the domain. Next, a domain representation might be used by a general model viewer in order to display the contents of a repository. Such a tool could be useful for developers of Memento–based assistants or administrators of MObject repositories. Finally, the domain representation will likely be used by the network repository to allow it to send MObjects to a connected assistant that it would likely later fetch in the process of MObject realization. A Memento–based MObject server equipped with this capability could have high performance without the need for programming specific to the domain it serves. Memento's network repository, which enables assistants to participate a client–server relationship, was described in Section 2.5.6.

3.6.2 Memento Facilitates Collaboration over Models

Together, Memento's first and second levels provide the essential support for collaboration over an arbitrary assistant's models. Memento provides this support via mechanisms that are independent of the the assistant's domain. Memento is the first system to support meaningful artifact–based collaboration in a domain–independent way. Memento is compared to other artifact–based collaboration systems in Section 5.1. Because of the difficulty of creating a collaborative assistant without an infrastructure such as Memento, it is expected that Memento will enable the construction of a wide range of assistants which otherwise would be too expensive or difficult to create and maintain.

Memento has the potential to enable a new class of assistants that would allow collaboration *across* domains. This capability is considered the ultimate goal of work on Memento. Multiple types of assistants can use a single shared repository and potentially share MObjects between them. Two levels of sharing are possible. MObjects may appear in the context of two or more separate domains. A single concept, for example, can be linked into models recognized by more than one domain. Through the normal MObject collaboration mechanisms of Memento's first level, changes made in one assistant would appear in another. In a deeper level of sharing, multiple assistants can share some or all of of their metamodels. In such a case, a higher level of meaning could be shared among disparate assistants.

3.6.3 Memento Facilitates Evolving Information Spaces

One way to think of the MObjects in a repository is as a collection of information that is shared by a group of users. Such a collection is sometimes called an *information space*. Often, it is the users' need to store and share information in a common space that drives the development of assistants used to manipulate it. An assistant, then, is only the means to accomplishing a user's goal and is not his primary concern.

An information space can evolve to become a valuable resource for a community of users. While the information space may grow in volume over time, it is rarely volume that makes such a resource valuable. Instead, the value is in the *richness* and *meaning* of the information. Richness and meaning are achieved primarily through adding detail and interconnections to information already present. Over time, these additions evolve from superfluous annotations to a core part of the information space.

Memento directly supports the processes of adding richness and meaning to information spaces. MObjects may contain arbitrary attributes which are not required by the MType system. Over time, new MObject subtypes may evolve which require the presence of certain of those previously unrequired attributes. MObjects having the required attributes can then have their types changed to the new subtype without ill effect to any context in which that MObject appears. A similar evolutionary process can occur in the conceptual graphs of an information space. Existing concepts can be linked by new relations which are not recognized by any metamodel. Recall that a metamodel characterizes the minimal set of connections that must be present for recognition and does not usually restrict the presence of additional connections. Some time later, assistant's metamodels may evolve to recognize and use the new interconnections. Often, the evolution of an assistant and its information space are intimately tied.

3.6.4 Memento Supports the Development of Assistant Applications

Memento embodies the problems of real–time collaboration, persistent storage, and meaning preservation in a single, reusable package. With a small amount of customization and a suitable user interface, fully functional, collaborative assistants can be easily constructed using Memento.

Memento's suitability as the basis for an arbitrary assistant is largely determined by two issues: model expressiveness and MObject granularity. Model expressiveness is the question of whether an arbitrary assistant's models can be represented in terms of Memento's model representation, that is, conceptual graphs with component concepts and relations constructed from typed, attributed collections (MObjects). While Memento's model representation scheme is quite general, it is conceivable that it will not satisfy the needs of all assistants. One potential source of difficulty is the fact that the type system over MObjects is a hierarchy and not a fully general type lattice. Chapter 4 describes several assistants that have been built using Memento. While these examples are small, in none of the cases was the underlying data model a source of difficulty. It may be that only through the experience of developing a large number of Memento–based assistants can the suitability of conceptual graph and MObject data models be fully determined.

The second issue of Memento's suitability for assistants concerns the granularity of information stored in MObjects. Granularity can have a big impact on the performance of an assistant. Consider, for example, the case of a collaborative writing tool. Should the most detailed concepts be a document, document section, paragraph, sentence, word, or letter? Clearly letters and words are probably too small to be entire concepts. The sheer volume of MObjects that such an assistant would have to manipulate would make such an assistant unusable. The larger granularities of document and document section are large enough to impede collaboration through locking conflicts. Because of the unusual needs of a collaborative writing tool, it may not be a good candidate for construction with Memento.

3.6.5 Implementation Status

Currently, all of the first–level Memento mechanisms described in Chapter 2 are implemented and working. The metamodel representation and model recognition algorithm are both implemented, as well. The `Model` class and metamodel unification are only partially implemented, hence their design has yet to be fully validated.

Chapter 4

Customizing Memento for Specific Domains: Experimental Results

Using Memento's first two layers, a developer can easily construct a third layer that is specific to a particular assistant's domain. This domain–specific layer is called the *domain representation*. The domain representation can be thought of as a portable, collaborative application programmer's interface (API) for the assistant's models. This chapter outlines the process of creating a domain representation and its part in the construction of a complete assistant. Next, case studies of six Memento–based assistants are presented. The assistants are not only example systems; they demonstrate the key capabilities of the Memento infrastructure outlined in Section 1.4. The chapter concludes with a discussion of how the assistants demonstrate each of the intended capabilities.

4.1 Domain–Specific Customization Methodology

A *domain representation* is a customization of Memento for a specific domain. A domain representation consists of four components:

- a set of `MObjectClass` instances that declaratively describe an assistant's constraints over its MType system;
- a set of `MObject` subclasses corresponding to each newly specified MType;
- a set of `ModelClass` instances that declaratively describe an assistant's constraints over its metamodels; and
- a set of `Model` subclasses corresponding to each of the newly specified metamodels.

These components are derived from an analysis of the target domain. Object–oriented analysis and design [10, 7] is one such analysis method.

The declarative `MObjectClass` and `ModelClass` instances describe the assistant's semantic constraints that Memento can directly enforce. An `MObjectClass` instance characterizes a domain–specific MType and its expected attributes. `MObjectClass` instances were described in Section 2.4.2. A `ModelClass` instance uses `MObjectClass` instances to describe a domain–specific metamodel and its expected interconnection constraints. A `ModelClass` instance acts as a placeholder for the associated metamodel, which is constructed at assistant run–time. Section 3.4 described the construction and use of `ModelClass` instances.

For each newly specified MType, the developer creates a corresponding `MObject` subclass. Similarly, for each newly specified metamodel, he creates a corresponding `Model` subclass. The primary purpose of these classes is to provide domain–specific convenience access and modification methods and domain–specific notification methods. In the case of the `MObject` subclasses, access and modification convenience methods allow the convenient manipulation of an MObject's required attributes. Such methods were described in Section 2.2. A developer can also customize `MObject` subclasses to give specialized notification of dynamic changes to the MObject's required attributes. The MObject change notification mechanism was described in Section 2.3.3. At the conceptual graph level, a `Model` subclass provides access and modification methods to the graph structure of models recognized by its associated metamodel. Domain–specific, high–level change notifications can also be given by such a subclass. The responsibilities of `Model` subclasses were described in Section 3.3.

The domain representation's `MObject` and `Model` subclasses can also embody *ad–hoc* semantic constraints. Such constraints are called ad–hoc because they are not directly enforced by Memento. Ad–hoc constraints are either enforced within an MObject by an `MObject` subclass or over an entire model by a `Model` subclass. Depending on the particular circumstances of the constraint, the developer may choose to either bar transactions that violate the constraint, or adapt the appropriate notification mechanism to actively enforce constraint maintenance. Memento's transaction mechanism, described in Section 2.2, is amenable to either strategy.

A complete assistant is built from Memento, an MObject repository, a domain representation, and an assistant–specific user interface. Figure 4.1 shows a typical layering of these components. When combined with Memento, an assistant's domain representation becomes a portable, collaborative application programmer's interface (API) for the assistant's models and domain. An MObject repository, which is usually based on a persistent store, allows for the sharing and storage of MObjects. MObject repositories were described in Section 2.5. Most often, an assistant's user interface component is constructed from an application framework, such as the Microsoft

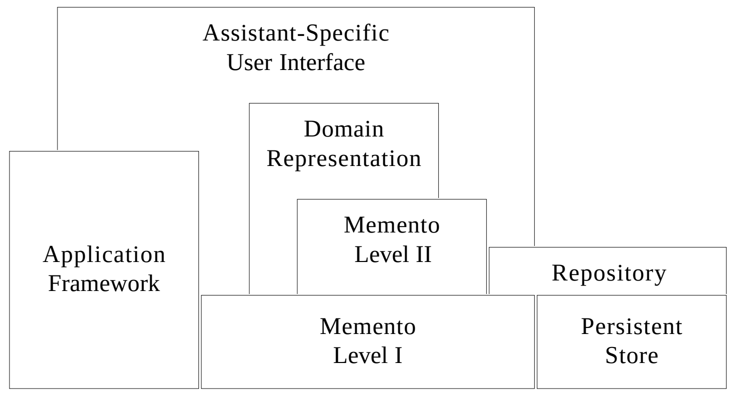

Figure 4.1 A typical assistant architecture layering

Foundation Class Library [81]. This is the case for many of the assistants described in this chapter. Application frameworks are discussed further in Section 5.3.

4.2 Assistant Case Studies

In this section, a number of Memento–based assistants are described. These assistants were developed by the author and the author's research group over the span of several years. The acknowledgments section credits the various people who have contributed to the development of these assistants.

4.2.1 "Virtual Reality" Walker

The Virtual Reality Walker (VRW) assistant is a simple, non–collaborative, modeler application inspired by the popular computer game "Myst" [12]. In the game, the user navigates around a computer–generated island and solves puzzles. For the most part, what the user sees is rendered images from various positions on the island. Transitions to other positions are suggested by portions of the image that the users sees (such as a stairway) and by changes in the mouse cursor as the mouse moves over that "hot region" of the screen (such as an arrow indicating the direction of potential travel). When the user clicks on the hot region, a new image is displayed from that new perspective. A carefully constructed set of interconnected images can give the user a real sense of navigating in a physical space. This simple model of interconnected views is the basis for the VRW assistant, a 32–bit Windows application built with

Figure 4.2 Screen snapshot of the Virtual Reality Walker assistant

the Microsoft Foundation Class Library [81]. VRW's images were generated from a digital camera in an office building's atrium and hallways. Figure 4.2 shows a screen snapshot of the VRW assistant showing a view down a hallway. Myst, however, builds on this idea in many ways including ambient sounds, sounds associated with transitions, animations, and saved state.

The VRW was built in several stages to demonstrate how Memento supports evolution of an assistant and its associated information space. The first version of VRW added a `Concept` subtype, `View`, for representing a view from a position in the virtual space and a `Relation` subtype, `Transition`, for the possible paths from a view. The `View` MType requires a single attribute for the displayed image. A non–required attribute was used to annotate the original image file name that was used to generate the concept's image attribute. For debugging purposes, this attribute is displayed in the lower pane of the assistant's main window. The `Transition` MType requires attributes describing a rectangular hot region of the view's image

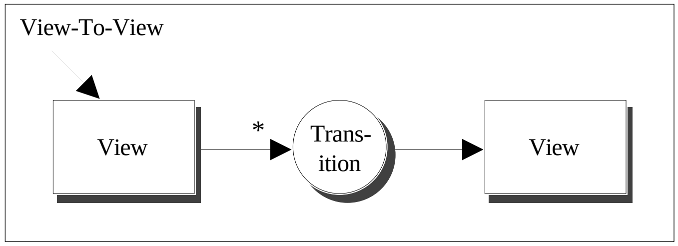

Figure 4.3 Initial metamodel for the Virtual Reality Walker assistant

that activates the transition. Additionally, another required attribute names a cursor to use in the hot region. Figure 4.3 shows the initial metamodel for the VRW assistant.

The first extension to the VRW assistant involved adding a `Transition` subtype, `TransSound`. The new MType requires an attribute for a sound and an attribute describing when, during the transition, the sound should be played. The assistant was modified to recognize the new subtype and play the sound, as appropriate. Several `Transitions` in the information space had the necessary attributes added and their MTypes changed to the more specific `TransSound` MType. Note that the original version of the VRW assistant remained able to use the same information space.

The next extension to the VRW assistant involved adding more interconnections to the information space. In particular, each view is associated with a concept representing a physical location; a single location would be associated with all views of different orientations from that position. A physical location is represented by a new `Concept` subtype, `Position`. A `Position` is associated with a `View` through a `Relation` subtype, `Vista`. Figure 4.4 shows three additional metamodels for this version of the assistant. The "Positioned View" metamodel allows navigation from a view to its unique position. The "Viewable Position" metamodel allows navigation from a position to all of its views. Next, the original "View–To–View" metamodel is modified to take advantage of the new "Positioned View" metamodel. Note that the reachable positions from a given position is embedded in the `Vista` and `Transition` relations. The final, "Reachable Positions" metamodel allows navigation from one position to all of those that are immediately navigable from a given position. Again, the first two versions of the VRW assistant remained able to use the updated information space.

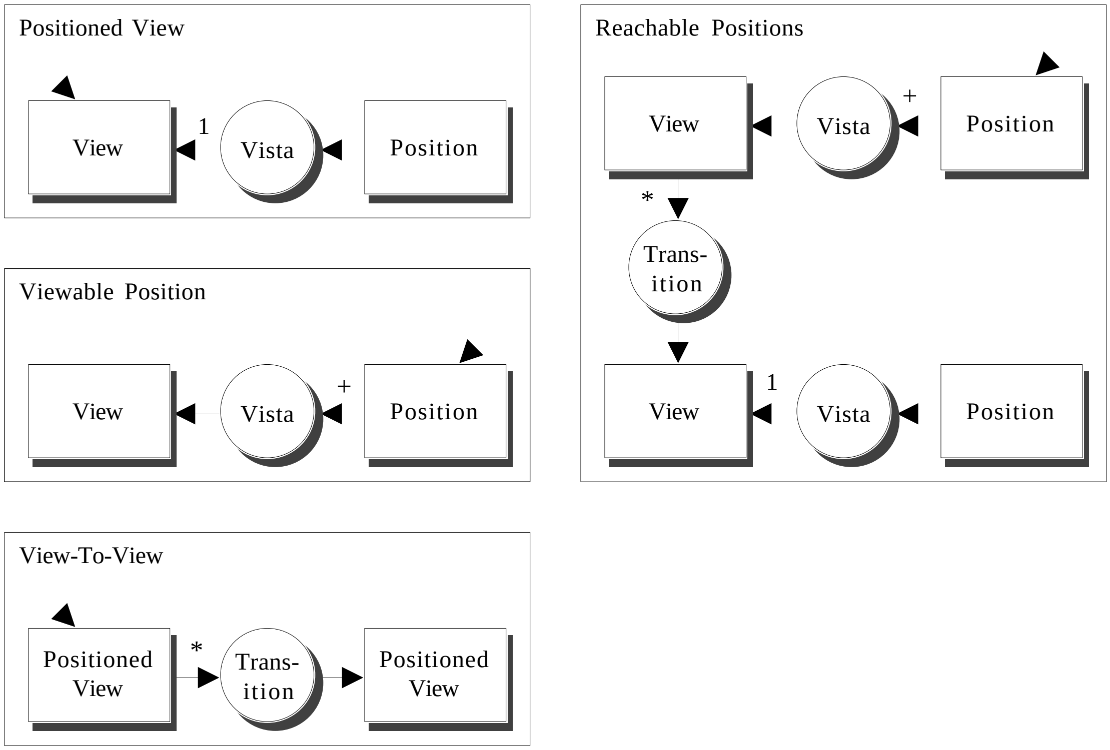

Figure 4.4 Second generation metamodels for the Virtual Reality Walker assistant

A final, hypothetical, extension to the VRW assistant might allow each position to be associated with its physical space, such as a room or hallway. The extended data model might allow the calculation of the sound amplitude of ambient noise source placed at some position in a room. The sound might not be heard if it originated in another room or if the listener was too distant from the source.

Thought was given to the idea of creating a simple multi–user dungeon (MUD) on the same, hypothetically extended information space as the VRW assistant. A "dungeon" visitor would have a command line interface that would allow him to navigate positions within the rooms of the dungeon (information space). Textual descriptions of what is "seen" from each position would be given as he went. If such an assistant were constructed, it might make sense to add explicit relations between navigable positions. The MUD would use metamodels involving `Positions` and Spaces and their interconnections, but would ignore Views and Transitions. Taken together, the MUD and the VRW assistant would provide two related views onto the same information space.

4.2.2 Elmo: Automatic Sharing of Screen Snapshots

Another simple assistant, called Elmo, was constructed which allows the automatic sharing of screen snapshots between users. Elmo is a 32–bit Windows application built with the Microsoft Foundation Class Library [81]. The tool attempts to address the need to share visual information during a telephone conference. Using Elmo, a user can specify a window on his screen to "broadcast" to others. The specified window can be the entire screen. Elmo takes a periodic snapshot of the window and stores it in the repository. Additionally, if the broadcaster's mouse cursor is in the window being broadcast, its position is saved to the repository. Other instances of Elmo viewing that image periodically see updates to their versions via the repository synchronization mechanism, described in Section 2.5.4. The broadcaster's mouse cursor position is reconstructed in the viewer overlaid on the broadcast image. For simplicity's sake, Elmo is restricted to broadcasting only one window at a time and viewing only one broadcast at a time. The user selects which of the available broadcasts to view from a menu. There is no restriction on the number of viewers for a broadcasted image. Figure 4.5 shows a screen snapshot of Elmo's viewing window.

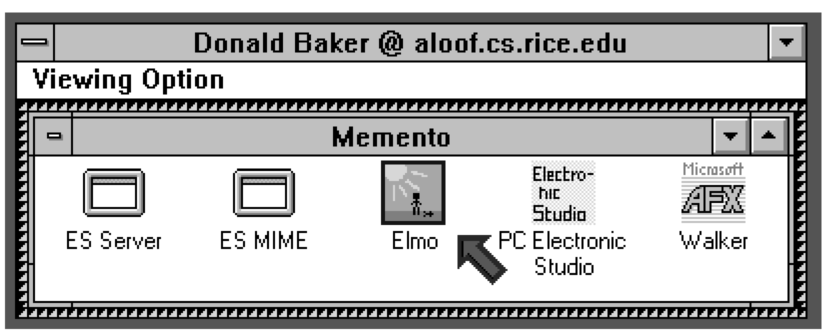

Figure 4.5 Screen snapshot of the Elmo assistant's viewing window including a rendition of the broadcaster's mouse cursor

Elmo's metamodel, shown in Figure 4.6, is quite simple. The metamodel simply associates a concept (ImageConcept) containing an image attribute with the Session

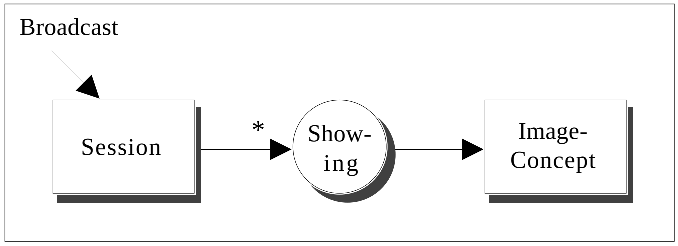

Figure 4.6 The Elmo assistant's metamodel

of the Elmo assistant generating the image. The `ImageConcept` MType requires a single attribute for the broadcast image. Every Memento–based assistant maintains a `Session` concept in every repository to which it connects. The `Session` MObject exists only during the time the assistant is connected to the repository. A `Session` enumerates information about the assistant executable, the machine on which its running, and the user that's using it. The broadcaster's cursor information is stored in the `Showing` relation that associates the `Session` with the `ImageConcept`.

Some comments about Elmo's performance are in order. Because color images can get quite large, the majority of time spent in Elmo is comparing, transmitting, dithering, or rendering images. When the broadcaster's image changes, the new image must be sent to the server after it has been dithered from 24 bits to 8 bits. The smaller image can thus be transmitted and rendered more quickly by the viewing instances. The broadcast process is largely limited by the sending instance. Broadcast updates of about three seconds have been determined to work well. The broadcast user can adjust his update time in one second increments. Note that if the image doesn't change from one update to the next, the repository is not updated. In this case, no image is transmitted over the network. If there is some change to the broadcast image, however, the entire image must be updated during the transaction, because of Memento's restriction of atomic attribute updates. Memento's atomic update strategy allows all MObject attributes to be treated in a uniform manner, regardless of their domain. In Elmo's case, Memento's generality prohibits domain–specific performance optimizations. Similar commercial products, most notably Farallon's Timbuktu, send partial image updates, rather than entire images. Such products sacrifice generality for better performance.

4.2.3 Electronic Studio

The Electronic Studio (ES) is Memento's flagship assistant. ES was heavily influenced by the Virtual Notebook System (VNS) [50]. The VNS is a mature commercial product which includes many features and mechanisms currently unaddressed by Memento, including user authentication, object permissions, client and server licensing, crash recovery, and intellectual property management. The main purpose of building ES was to create a shared, general information space, like that of the VNS, while generalizing the VNS's data model. ES will be compared and contrasted to the VNS throughout this discussion. Like the VRW assistant discussed above, ES was developed in several stages. ES is a 32–bit Windows application built with the Microsoft Foundation Class Library [81].

The Electronic Studio assistant's information space contains a set of interlinked pages. Each page shows a set of objects arranged in a two dimensional layout. The initial version of ES allowed three different object types on the page: a text document, an image, or a link to another page. Text can be imported from the file system, the clipboard, or edited in place. Images can be imported from the file system, the clipboard, or created by taking snapshots of portions of the screen.

The VNS allows much the same treatment of the three object types, but differs in other ways. The VNS supports a fourth object: an action. Actions allow platform–dependent scripts to be executed by the client application. Because action scripts often rely on the environment in which they are run and because of users' difficulty with them, they were not included in ES. The VNS organizes pages into notebooks, and notebooks into libraries. There is an implied containment in this hierarchy. A library contains its notebooks, a notebook contains its pages, and a page contains its objects. In contrast, objects on an ES page are associated by reference. The same object can appear multiple times on the same page or on several different pages. In each case the placement and the sizing of the object can differ. The superstructure of notebooks and libraries in the VNS is replaced by a different mechanism in ES, described later.

The next version of the Electronic Studio assistant added two more potential objects to a page: URLs and documents. Both are represented on the page by a rectangular block of text and an icon suggesting its nature. Both objects make use of a mechanism added to this version that allows ES to manipulate other applications on the client machine. A URL object holds the universal resource locator (URL), which is a pointer to a resource in the World Wide Web. Clicking on a URL object's icon launches a WWW browser window to that URL. ES has a menu item for capturing the URL of a currently showing WWW resource in the active WWW browser window.

A document object allows a binary representation of a file to be placed into the shared information space. Activating the document object, by clicking on the icon, begins a document editing or viewing session, which does not end until the activating page is closed.[18] At the beginning of the session, the document fetched by ES, is placed into the local file system, and the associated application is launched on that file. The association between documents and applications is part of the MIME information explicitly added (once) to a repository's MType system, as described in Section 2.4.1. If the user is the first to launch the document, he is allowed to edit a copy of the document while it is locked from modification by others. Document locking makes use of the application–level locking mechanism, described in Section 2.2.3. At the end of the session, the user is reminded to save his work back to the file, the file is saved back to the repository, and the MObject is unlocked. If another user has a document locked, a read–only version of it is copied to the local file system for viewing by the user. In this case, the document is not copied back to the repository at session end.

The final extension to the Electronic Studio assistant adds a sixth object type to a page: a hierarchical grouping, called a *suite*. A suite can be thought of as an outline in which each entry is either text, a URL object, a link to an ES page, or a document object. As with pages, a suite only refers to its objects, so they may be shared in multiple contexts. A suite can serve as a table of contents over pages in much the same way that the VNS's notebooks and libraries organize their pages. Unlike the VNS's notebooks and libraries, however, multiple suites can serve as multiple simultaneous organizations over the same set of pages. Suites are also the means by which queries over documents are invoked. See Section 4.2.5 for a discussion of this mechanism.

Figure 4.7 shows a page of the ES assistant showing a text object, an image, a link to another ES page, a suite, a URL object, and a document object. The suite contains some of the same objects. The cross–hair in the lower right corner of the page marks the insertion point for the next object to be created or pasted.

Figure 4.8 shows the metamodels for the ES assistant. An "ESPage" is a `PageConcept` connected via `PageElements` to a set of objects appearing on it. The `PageConcept` MType requires attributes for the page's title, size, and color. The `PageElement`

[18]Because of limitations of the Windows operating system, ES cannot determine when the launched application has terminated. A forced termination point is thus required.

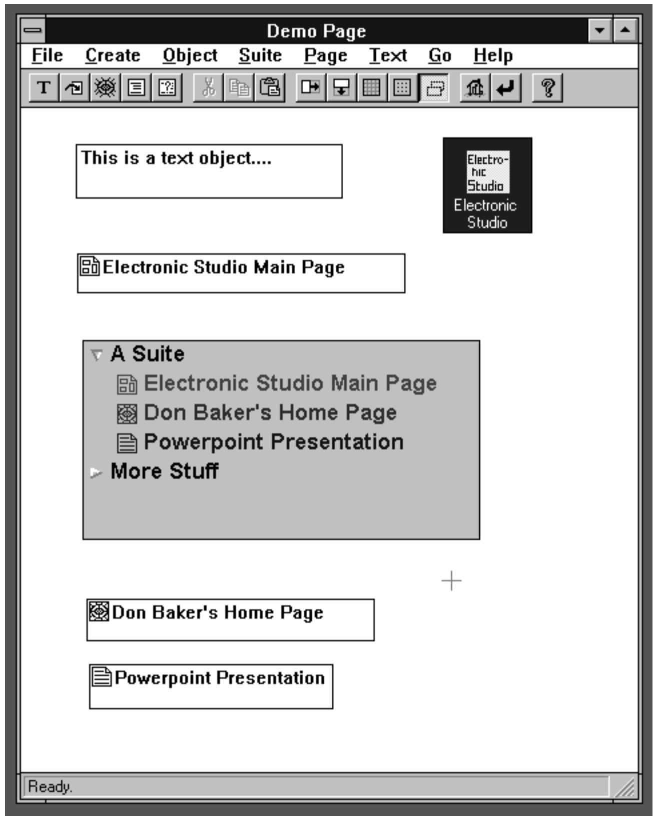

Figure 4.7 A screen snapshot of the Electronic Studio assistant showing a page with the various object types

MType records the position, size, label, and color of the object in the context of the page. What may not be obvious from the metamodels is that an object appearing in the context of one ESPage can appear in the context of other ESPages. An "ESSuite" is a `SuiteConcept` connected via `SuiteElements` to objects of the four possible object types that have a textual representation. The `SuiteElement` MType requires an attribute for the outline heading label. For efficiency reasons, the hierarchical structure of the outline is stored as a single attribute of the suite concept, but this decision violates the assumption that a concept is independent of its relations. A better representation is needed that would allow the hierarchy to be stored in a distributed fashion over the `SuiteElements` in such a way that few of them would be modified by common outline edits.

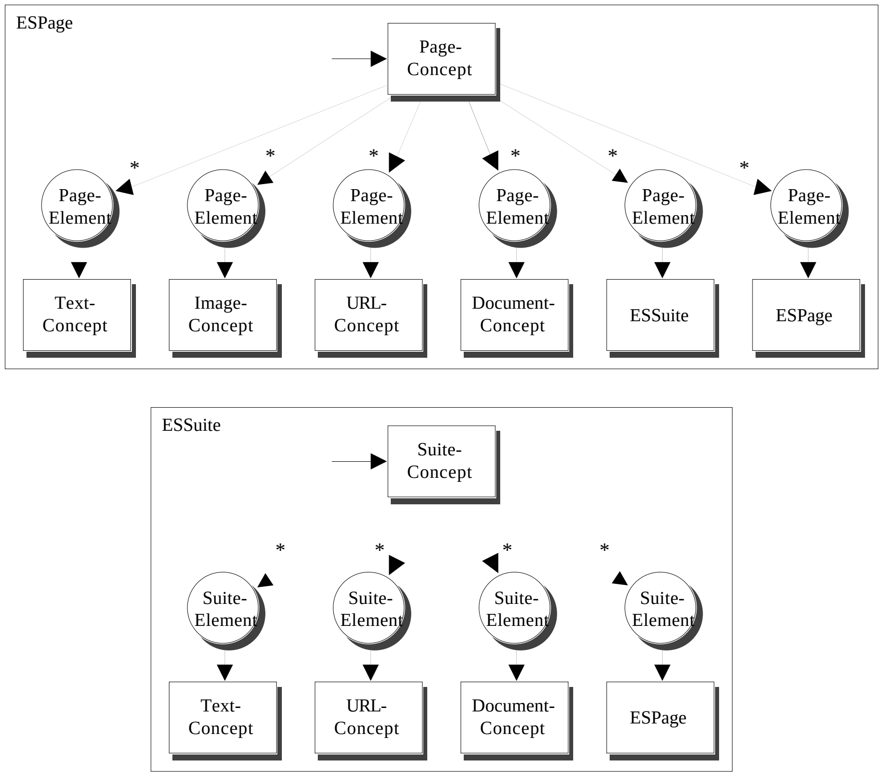

Figure 4.8 The Electronic Studio assistant's metamodels

4.2.4 Electronic Studio WWW Gateway Agent

The Electronic Studio WWW Gateway agent publishes the ES information space to the World Wide Web (WWW). The agent is composed of two parts: a Perl script and the agent service. The Perl script is registered with a WWW server, usually by placing it in the server's CGI–bin directory. A WWW browser can trigger the script, if the URL has the appropriate path. The script connects to the agent service, a demon process that accepts requests on a known port. The script passes the URL to the agent service which generates a HTML document for the corresponding ES page. The output HTML is sent back via the WWW server to the browser that generated the request.

The agent service generates URLs for any Electronic Studio object that can be fetched via the gateway. Most often, a URL corresponds to a page, but images and documents must have their own URLs, as well. For example, images are never part of an HTML document; they are fetched separately during a WWW browser's rendering process. For a document, the agent service sends its binary representation with a header based on the document's MIME information. Recall from Section 2.4.1, that document MIME types are an explicit part of the MType system. The ES information space is most often entered via an empty URL that maps to the main Electronic Studio page.

Because HTML does not allow arbitrary placement of items on a page, the two dimensional layout of an Electronic Studio page must be linearized by the agent service. A top–down, left–right linearization of the items on an ES page produces a reasonable approximation of the original. For those items with a textual representation, that text is output. Images are scaled to the size in which they appear on the ES page. Suites are output as tables. URLs, documents, and ES pages become WWW links to their destination.

4.2.5 INQUERY Document Indexing Agent

Another Electronic Studio agent adds a query capability over documents contained in the ES information space. The INQUERY Document Indexing Agent is built upon the INQUERY full–text information retrieval system [15], developed at the University of Massachusetts. The agent maintains indices for all documents in the information space by watching for newly created documents, document modifications, and document destructions, and invoking INQUERY to index documents, as needed.

Queries and their results are exchanged via suite objects in the ES information space. The Electronic Studio assistant can create a suite object on a page that represents a query for the agent to process. Information about the query is stored in several attributes of a suite, including the query string, the type of the query, and its state. Currently, the only query type supported is a one–time query, but periodic and continuous queries have been considered for future development. The query's state includes *pending*, *running*, and *done*. The agent watches for suite objects with attached queries, invokes INQUERY to determine the set of documents matching the query, determines the ES pages that refer to those documents, and populates the suite with an outline representing the results. The top level of the outline lists the matched documents with the referring ES pages appearing underneath. After a one–time query has been completed, the user can modify the generated outline as usual.

While the INQUERY Document Indexing Agent is complex, only minor modification to the Electronic Studio assistant was required to make use of it. The agent can run on a machine with computational and space resources necessary for maintaining the full index. The ES assistant is then endowed with a powerful search capability without requiring that the capability be part of the application. The agent and the ES assistant interact at a site in the shared information space. After the suite's creation, no further special interaction is required by the assistant. Through the normal Memento collaboration mechanisms, the user can watch as the suite is populated with query results. Alternatively, he may close the page containing the suite and later return to see the results of the query.

4.3 Capabilities Demonstrated

Section 1.4 set forth several capabilities that would be demonstrated by the assistants constructed with Memento. In particular, the assistants should demonstrate:

- human–problem domain communication,
- support for collaboration over models,
- support for the evolution of the assistant and its associated information space, and
- support for information agents.

. To summarize the work described in this chapter, each of these capabilities is now discussed in turn.

Human–problem domain communication is best demonstrated by the Electronic Studio assistant. Human–problem domain communication was first described in Section 1.3.2. Memento's mechanisms for human–problem domain communication were described in Section 3.5. The assistant provides a view onto domain–specific models representing ES pages in a shared information space. The models include pages, suites, and their associated objects. The user interface allows the straightforward manipulation of the models, such as dragging objects to change their position within the context of their page. The ES user interface component takes full advantage of the MObject notification mechanism, described in Section 2.3 to display incremental changes to the models. The ES domain is communicated through the assistant by the selective disabling of user interface capabilities made inappropriate through dynamic modification to the assistant's metamodels.

The Electronic Studio assistant also aptly demonstrates how Memento supports collaboration over models. Section 2.8 described Memento's mechanisms for collaboration over MObjects. These mechanisms include: persistent MObject storage in repositories, coherent modification of MObject's via Memento's transaction mechanism, repository–initiated transactions during the repository synchronization process, and user interface updating via Memento's MObject change notification mechanism. Adding to the mechanisms of Memento's first level, the second level ensures that the semantic constraints of an assistant's domain are not modified during transactions. Section 3.6.2 described how the combination of Memento's two layers support artifact–based collaboration over domain–specific models.

The Virtual Reality Walker (VRW) assistant demonstrates Memento's support for evolving assistants and their associated information spaces. The VRW assistant underwent several incremental changes. Each change added richness and complexity to its information domain as described in Section 3.6.3. The Electronic Studio assistant was also developed in an incremental fashion and its information space evolved over time.

Finally, Memento supports the needs of information agents, as was demonstrated by the INQUERY Document Indexing Agent and the Electronic Studio WWW Gateway agent. The first monitors changes in the information space and makes specific changes to it, in response. The agent acts as a collaborator over the information space. The second agent performs a translation from one information space to another (the World Wide Web).

Chapter 5

Context and Contributions

Memento is an infrastructure to support the development of assistant applications. As such, Memento is a contribution to the broad area of *augmentation*, or the use of computers to enhance human intellectual capabilities. Early contributions to this field include the *Memex* system proposed by Vannevar Bush in 1945 [16] and Doug Engelbart's ground–breaking NLS/Augment system [36] in 1968. Both contributions have made a profound impact on computer science and have served as the premier systems in several subsequently spawned fields of research, most notably that of hypertext and hypermedia. Today, important subfields of augmentation include human–computer interaction and computer–supported cooperative work (CSCW). Cognitive science is another important field that falls partially under the umbrella of augmentation. The conceptualization and implementation of Memento draws from research in these three fields. In particular, Memento aids in the construction of user interfaces at the semantic level of communication, it is a contribution to the area of CSCW called *artifact–based collaboration*, and Memento employs a knowledge representation scheme with a basis in cognitive science. As a software infrastructure, Memento attempts to address many of the issues software developers might face over the lifetime of an assistant or suite of related assistants. As such, Memento addresses many software engineering issues, including software reuse and evolution of an assistant's code. The first four sections of this chapter place Memento in the context of the fields of artifact–based collaboration, human–computer interaction, software engineering, and knowledge representation. Note that these sections are interrelated.

Memento also touches on several other important, but peripheral areas that are not discussed here. These areas include agent technologies, graph pattern matching algorithms, and database technologies, such as semantic database models, extended transaction mechanisms, schema migration and evolution, and federated and distributed databases. Memento's relation to object–oriented databases was discussed in Section 2.7. The chapter concludes with a summary of the contributions of this dissertation and a section on possible future work on Memento.

5.1 Artifact–Based Collaboration

Computer supported cooperative work (CSCW) systems, sometimes called *groupware*, seek to help people work together through the medium of computers. CSCW is a broad field that includes computer conferencing, multi–user dungeons (MUDs), blackboard technologies, collaborative document authoring, workflow applications, and electronic mail [35]. Unfortunately, because of the commercial importance of CSCW, many important and useful systems are not described in the academic literature. First–hand study of such proprietary systems is either difficult or prohibitively expensive. The few commercial systems that are available to the author will be mentioned here.

CSCW systems are often classified as to their ability to support collaboration across space and/or time [35, 96]. In one extreme of the space dimension, an electronic classroom might require the co-location of the participants. In the other extreme, *distributed* systems, such as electronic mail, place few constraints on the users' physical locations. The time dimension describing CSCW systems varies from *synchronous*, meaning all participants interact at the same time, to *asynchronous*, where the collaboration effectively bounces between participants over time.

Another way to classify CSCW systems is by the extent the system is *artifact–based*. Artifact–based collaboration systems address a core CSCW problem of sharing an information space [4, 60, 92]. In artifact–based collaboration, *information artifacts*[19] are shared among the participants and are usually the focus of the collaboration. Note that whether a system is artifact–based is, to some extent, independent of the space and time dimensions. Only asynchronous collaboration requires that the collaboration be artifact–based so that state can be preserved between users' sessions. Synchronous artifact–based collaboration systems are often referred to as *real–time* because updates occur immediately. Memento supports real–time artifact–based collaboration over an assistant's models. The remainder of this section focuses primarily on systems and issues related to real–time artifact–based collaboration, because they are most relevant to Memento's capabilities. Specifically, areas of message systems, workflow, or digital libraries will not be covered. After a brief discussion of the classes of real–time artifact–based collaboration systems, the toolkits that support these kinds of applications are then described. The section closes with a discussion

[19]This term was defined at the beginning of Chapter 2.

of organizational memory, the ultimate purpose of many artifact–based collaborative systems.

5.1.1 Systems for Real–Time Artifact–Based Collaboration

Real–time artifact–based collaboration systems generally fall into the subcategories of domain–specific systems, collaborative document editors, hypertext and hypermedia systems, and virtual work spaces. Each of theses subcategories is now visited, in turn, with a discussion of how Memento relates to each.

Domain–Specific Systems

There are many artifact–based systems that address specific information or problem domains. Domain–specific systems may be well suited to their specific problem area, but often are useless outside that area. A representative list of systems in this diverse category include:

- Visual Scheduler [6], a group calendar and scheduling application;
- BIBDB [83], a collaborative system for bibliographic data;
- XNetwork [92], a collaborative system for designing network layouts;
- ICICLE [14] for code inspection and annotation;
- IBIS/gIBIS [23, 24, 110] and SIBYL [74], collaborative systems for group design deliberation;
- Colab [104], an early system to support group meetings; and
- systems for group decision making, surveyed in [70, 29].

Domain–specific artifact-based collaboration systems such as these are considered assistants because they each have an understanding of their specific domain. These systems can be implemented using the domain–independent Memento infrastructure. In doing so, a bulk responsibility of domain–specific artifact–based collaboration in each assistant could be delegated to the resuable Memento infrastructure.

Collaborative Document Editing

Collaborative document editing systems assist with the important task of group authoring. Representative systems include GROVE [35], Quilt [45], and the collaborative editor toolkit, DistEdit [67]. As mentioned in Section 3.6.4, Memento is probably not suitable for the development of collaborative document editing systems because of the fine information granularity required by such systems.

Collaborative Hypermedia

Hypertext systems support "non-sequential" reading and writing by allowing users to traverse links that interconnect sections in a web of interrelated documents. When this idea is extended to other media, such as images, video, and sound, it is called *hypermedia*. Hypermedia has grown in importance due to its natural flexibility. While the focus in hypermedia is usually on the reader of static hypermedia webs, many systems support authoring. A subset of those systems support the real–time collaborative construction of hypermedia webs. These systems include NLS/Augment [36] from the Stanford Research Institute, Xerox's NoteCards [57, 59], Brown University's Intermedia [80, 73], the University of North Carolina's Artifact Based Collaboration (ABC) System [60, 102], and SEPIA [105] from the Integrated Publication and Information Systems Institute at the German National Research Center for Information Technology (GMD). Note that all of these systems, with perhaps the exception of NLS/Augment, are also considered toolkits for supporting artifact–based collaboration. CSCW toolkits will be discussed in Section 5.1.2. Collaborative hypermedia systems for specific domains include Baylor College of Medicine's Virtual Notebook System [50][20], whose hyperlinked documents are pages organized into notebooks, and gIBIS [110, 23, 24], a collaborative hypertext tool specialized for team design deliberation developed at the Microelectronics and Computer Technology Corporation (MCC). The relation between Memento and collaborative hypermedia is discussed later, in Section 5.1.2 where Memento is compared with various hypermedia toolkits.

Virtual Workspaces

Another class of real–time artifact–based systems is virtual workspaces. Two key components of virtual workspaces are a sense of locale and an awareness of others also present in one's locale. Typically, the locales are portrayed as interconnected rooms where the participants in each room have an awareness of the others present. Representative systems include WORLDS [47, 108] from the University of Illinois at Urbana–Champaign and Xerox's Jupiter multi-user dungeon (MUD) [27]. Studies

[20] The Virtual Notebook System is now a commercial product marketed by the Forefront Group.

of these systems focus on the social interaction that occurs in the workspace [46]. Virtual workspace systems are artifact–based in the sense that the presence of locales and participants are shared artifacts. In some systems, participants can bring information artifacts, such as documents, to be shared with other participants in their locale. The more important aspect of virtual workspaces, however, is the synchronous collaboration that occurs in the space, usually through some form of video conferencing. As the Elmo application in Chapter 4 demonstrated, Memento can support the artifact–based component of a virtual workspace system.

5.1.2 CSCW Toolkits

Among other things, Memento is a toolkit for the development of collaborative applications. This section describes how Memento relates to other toolkits that support collaboration. The classification of computer supported cooperative work (CSCW) toolkits developed by Dewan [30] categorizes Memento as a *shared object system*. Shared object CSCW toolkits are well suited for use with assistants; because they are artifact–based, they can provide for real–time updates without embodying assumptions about the user interfaces of their applications. This last quality is essential if different versions of an assistant are allowed to manipulate the same information space.

By contrast, many other toolkit classes focus on the sharing of user interface components that can appear to multiple users. In such systems, sometimes called *what–you–see–is–what–I–see* or *WYSIWIS*, shared user interface components are replicated between applications. In many of these systems, applications have little or no semantic knowledge of their data, but simply act to coordinate the shared input stream and maintain view consistency. Important toolkits that address the sharing of user interface components include GroupKit[94], Suite [31], COAST [97], DistView [91], and Rendezvous [89]. Often, the focus of these systems is on the process of collaboration, specifically with regard to coordination of actions, and not on the information artifacts being manipulated. As currently developed, Memento does not provide mechanisms to aid in the process of collaboration, however, such mechanisms could be built as a layer upon Memento. There is often a trade–off in such toolkits between being general purpose and exploiting the semantics of the collaborative application to optimize performance [33]. While Memento's focus is on domain–independence, Memento may be able to make use of an assistant's explicit semantic information to improve performance. This possibility was discussed in Section 3.6.1.

Memento's focus is on meaningful information artifacts, called *models*, that are manipulated by Memento–based assistants. Key goals of Memento include:

- real-time collaboration over graph–structured information (in the form of conceptual graphs),
- employment of explicit semantic information,
- support for assistant and information space evolution, and
- support for information agents.

Memento is the only toolkit that addresses all four of these issues. The following four sections describe these issues in more detail.

Collaboration over Graph–Structured Information

As mentioned above, hypermedia systems are an important class of graph–based collaboration systems. Collaborative hypermedia toolkits, therefore, must facilitate collaboration over graph–structured information. Recall from Section 3.1 that Memento uses a conceptual graph data model to represent domain–specific models. The subjects of how Memento relates to collaborative hypermedia toolkits in general, how Memento compares with one such toolkit, and finally, how Memento compares with other non–hypermedia toolkits that deal with graph structures are now discussed.

While both collaborative hypermedia toolkits and Memento deal with real–time collaboration over graph structures, Memento's graph structures are at a somewhat lower level than those of hypermedia toolkits. Memento's graphs are composed of atomic data items, whereas hypertext links may anchor within portions of documents that are the nodes of a hypermedia graph. It would be possible to build a collaborative hypermedia application using Memento, if one used the strategy employed by SEPIA [105]. SEPIA only allows interlinking between atomic document components, but allows components to be aggregated into compound documents.

Important hypermedia toolkits include Intermedia [80, 73], NoteCards [57, 59], ABC [102, 60, 98], and DeVise Hypermedia System [55]. Of the mentioned systems, ABC's foundational component, the Distributed Graph Server [98], is most similar to Memento. Within it, primitive objects include attributed nodes and attributed links. Both nodes and links may additionally have "content," which is often a raw data file. Subgraphs of connected nodes and links are first class objects in the graph server.

Subgraphs can also be stored as the "content" of a link or node. The net effect is that of containment or partitioning of a graph. For performance reasons, links are stored separately depending on whether they are within a subgraph or between subgraphs. Finally, the distributed graph server will enforce rudimentary constraints over the structure of subgraphs, such as that it be a list, a tree, acyclic connected, or connected. Through the use of multiple metamodels, Memento can support nearly the same notions of partitioning and containment as the distributed graph server. Additionally, Memento's metamodels aid in the enforcement of semantic constraints over its graphs—a more powerful notion than structural constraints. ABC's graph server is more sophisticated, architecturally, as it is truly distributed. Memento's architecture is based on an extended client–server model using the network repository as the bridge between an assistant playing the role of a server to a set of assistants playing the role of clients.

Other important systems to support collaboration over graph structures include MIT's Oval [76, 77], based on the earlier ObjectLens [72] and the University of Illinois at Urbana–Champaign's WORLDS [47, 108], which uses graph structures to specify activities and workflows.

Oval, which stands for *objects*, *views*, *agents* and *links*, is a collaborative environment which supports the *end–user* development of applications. Objects in Oval are typed collections of fields and actions. The fields hold untyped data values and the actions operate on the specified fields of the object. Object types form a hierarchy where objects of one type inherit the fields and actions of its supertype. Links represent relationships between objects in Oval, but they do not themselves contain data or a type. Views, which can be customized by end–users, display objects and links. Oval's agent are examined later in this chapter. Like Memento, Oval uses graph structures for a domain–independent basis for the construction of domain–specific applications. Oval focuses on end–user customizability and sacrifices portability and robustness. Memento takes the opposite approach. End users cannot typically customize Memento–based applications, but Memento can be the basis for portable domain representations and robust applications. The Electronic Studio assistant and related Electronic Studio WWW Gateway Agent, described in Chapter 4, demonstrated potability and the potential for robust, real–world applications.

WORLDS is a collaboration environment based on the specification language *Introspect* [108]. Unlike most artifact–based systems, WORLDS focuses on the dynamic and situated collaboration in the context of social worlds. WORLDS uses "spatial metaphor to firmly situate actions and interactions in an integrated work context."[21] WORLDS represents the processes and interactions that occur within and among locales via dynamic graph structures along which information flows. WORLDS can be considered a generalization of the workflow concept found in many commercial collaboration products, most notably DEC's Linkworks [32]. Memento currently provides no direct support for representing actions and flows. However, Memento could support such artifact–based extensions. Information flows and actions could be expressed in a Memento–based domain representation. A Memento–based information agent could act as an engine for dynamically executing the actions specified within flows.

Explicit Semantic Information

While most CSCW toolkits allow for the construction of domain–specific applications, few allow specification and use of the semantic information associated with the domain. Systems that employ explicit semantic information are more powerful than those that do not. Memento uses explicit semantic information to preserve the meaning of its information artifacts, or models. Two contrasting approaches for dealing with explicit semantic information could be called the *programming approach* and the *data approach*.

In the *programming approach*, semantic information is captured in the meta-classes of an object–oriented programming language. For example, WORLDS [47, 108] uses meta–specifications in the programming language, *Introspect* [108], to allow the dynamic creation and modification of workflows. Similarly, the VODAK Model Language [66], which is the basis for the collaborative hypermedia system SEPIA [105], uses an open object–oriented data model, based on the concept of meta-classes, to embody semantic information. While the programming approach provides for powerful specification mechanisms, it is often at odds with the goals of collaboration and the evolution of applications. Because the meta–level specifications are so closely tied to their programming languages, they cannot be easily shared between applications. As with all strictly programmed systems, care must be taken to prevent incompatible versions of the same application from using the same information space.

In the *data approach*, semantic information is represented as data. The data approach is less powerful because data representations of semantic information are

[21] [108] p. 56.

less expressive than programming languages. Memento follows the data approach; Memento's metamodels and MType system are data. In contrast to the programming approach, semantic specifications can be used in the collaboration process to ensure meaningful interchange. This issue was discussed in Section 2.0.2. Additionally, the data approach is more amenable to evolutionary change, as described Memento in Section 3.6.3.

Another system that follows the data approach is SHADE [79], a collaborative engineering design system developed at the Stanford Knowledge Systems Laboratory. SHADE uses the Knowledge Interchange Format (KIF) to represent the semantics of the information to be shared between cooperating components. KIF–based information appears as LISP–like statements. KIF is a much more expressive representation scheme than the conceptual graph structures used by Memento. On the other hand, KIF information is not amenable to the graph pattern matching and evolution schemes embodied in Memento.

Support for Evolution

Section 3.6.3 described how Memento supports the evolution of both the assistants and their associated *information spaces*. Memento is one of only a few artifact–based collaborative systems to address these issues. EGRET, HOS, and CoVer are three other such systems.

The EGRET Framework from the University of Hawaii addresses artifact–based collaboration in the context of evolutionary change [62, 63]. EGRET provides mechanisms for evolution similar to those of Memento. EGRET's developer, Philip Johnson, coined the term *exploratory collaboration* to describe situations in which evolution is anticipated. Frederick Brooks argued, however, that *growing* programs is a more apt metaphor of software development than *building* programs [13]. Section 1.1.2 identifies the user interface component as a significant source of evolutionary change in assistant applications. Because evolution of any software system is an important part of its life cycle, support for assistant evolution is a goal of Memento.

The EGRET data model consists of *nodes*, *schemata*, *links*, and *layers*. A *node* contains a set of fields for data values. *Schemata* form a hierarchical type system over nodes by specifying validity functions for possible field values. Schema "subtypes" may add or remove fields from its "supertype." A node is considered to conform to a schema's type if it satisfies all the schema's validity functions.[22] A *link* is a typed pointer from a field to a node. Links contain no data, but they do have an associated type that specifies a validity function over domain and range node schemata. A node's field may contain a single link or a set of links conforming to the link's type. A *layer* is a partitioning of the data space into potentially overlapping sets of schemata, node instances, and links. Layers serve both as a name-space mechanism and as a versioning mechanism over all components of the information space.

Fundamental to the EGRET model is the notion that all nodes and links may diverge arbitrarily from their schemata. Such divergence facilitates the evolutionary process Johnson calls *Ex–Con*, for *exploration and consolidation*. In the exploration phase, alternate schema and node and link instances diverge from those in the consensus layer. If the exploration is deemed successful, a consolidation phase migrates the primary portion of the database to a layer that describes the new consensus schemata and node and link instances.

There are three fundamental differences between EGRET and Memento. First, Memento has a richer data model than EGRET. Memento's relations can have attributes. Required attributes on relations are specified by the MType system. Memento's relations can can be n-ary rather than just binary. Finally, Memento provides metamodels for the specification of graph types. Memento's richer data model allows it to express a wider variety of domains. The next between EGRET and Memento is that Memento preserves the meaning of its information by requiring that modifications done in the context of an assistant's metamodels do not violate their embodied semantic constraints. As a result, Memento never allows large portions of the information space to become invalidated. In contrast, EGRET's constraints need not be heeded during manipulation of either instances or schemata. Because of these differences, EGRET cannot claim a high level meaning fidelity during its manipulations. Finally, Memento facilitates evolution by favoring changes that add richness and interconnectivity to the information space. Specifically, the operations of adding unrequired attributes, adding new types, changing the type of an MObject to a more specific type, and adding relations of new types are cannot violate the semantic constraints of any domain and are therefore always allowable.

The Hypertext Object Substrate (HOS) [99], developed by Shipman, provides mechanisms to aid in the evolutionary process of adding structure and detail to the

[22] Note that EGRET's type system does not have the property that a node conforming to a schema must also conform to its parent schema.

information space, called *incremental formalization*. Through incremental formalization, users add attributes to objects and extend the type system over objects to expect those attributes. (HOS has a data model similar to that of Oval, described earlier.) HOS encourages this process by suggesting attributes to users and by making the type system less restrictive (and presumably more user friendly). Shipman and Reeves have studied how information spaces have evolved through the process of incremental formalization in various user communities. Memento allows for the evolution of an information space through the addition of non–required attributes, via domain–specific extensions to the MType system, and by the changing of an MObject's type to a more specific one, thus requiring more attributes. Memento does not, however, provide user–level mechanisms to encourage this process. Additionally, Memento provides mechanisms for the evolution over the graph structures of an assistant's models. HOS does not address evolutionary mechanisms involving its links.

A well–known way for dealing with the evolution of an information space is by versioning the contained information. However, few collaborative systems have addressed this issue. CoVer [56] is a hypermedia versioning server that is part of the SEPIA [105] hypermedia toolkit, mentioned earlier. Memento does not currently address the issue of versioning, but such support would be quite useful to assistant developers. Memento will likely include mechanisms for MObject versioning in the future.

Support for Information Agents

Information agents are semi–autonomous programs that can act on an information space and react to changes within it. Section 1.1.1 identified agents as an important class of assistants. Memento supports information agents primarily through its MObject change notification mechanism, described in Section 2.3. Two Memento–based information agents were demonstrated in Chapter 4. Other collaborative systems, including Oval [76, 77], HOS [99], Linkworks [32], WORLDS [47, 108] (all mentioned earlier) provide support for agents. Oval even supports the end–user construction of information agents. One key difference between Memento and these systems, however, is that like any other Memento–based assistant, a Memento–based agent operates only on the models from its specified domain. Additionally, any such modifications an agent might make must adhere to the semantic constraints of the specified domain. In contrast, agents of other systems are not constrained by the types of modifications they can make. The potential of Memento–based information agents has only begun being explored. Memento–based information agents have the ability to act as a full collaborator over an information space shared by a user community. The potential power of this capability was aptly demonstrated by the Memento–based INQUERY Document Indexing Agent, described in Section 4.2.5.

5.1.3 Organizational Memory

One of the more important uses for artifact–based collaborative systems is for the construction of the shared knowledge of a group or organization, called an *organizational memory*. Because of their flexibility, hypermedia systems most often serve as the foundation for organizational memories. Indeed, the recent growth in corporate intranets is driven mostly by the need for organizational memories and the applicability of hypermedia–based World Wide Web technology to meet that need.

Two important uses of organizational memories are for shared design environments and for computer supported collaborative learning (CSCL). Examples of design environments are XNetwork [92] for network designs and IBIS/gIBIS [23, 24, 110] for capturing design deliberation. Both systems have been mentioned earlier. Another domain in which design has been extensively studied is software engineering design environments [42, 107]. Computer supported collaborative learning (CSCL) uses artifact–based collaboration for the end of education. CSCL systems are sometimes called *collaboratories* [69]. Not surprisingly, the focus of these systems is on their effectiveness as a teaching tool and the experience of the learner using them [109].

Two important issues in organizational memories are how to capture and use semantic information and how such memories grow and evolve over time. As mentioned earlier, Memento addresses both of these issues. Capturing semantic information allows the organizational memory to have more meaning and richness. It also enables the use of more powerful software tools, such as *information agents*. Agents can provide critical assistance to designers [34]. Semantic information can be incorporated either through direct programming of domain–specific collaborative tools, or through the use of a domain–independent knowledge representation scheme. Memento uses the latter approach. Knowledge representation schemes are discussed in Section 5.4.

Issues arising in the evolution of group memory include:

- how organizational memories must undergo periodic reorganizations, called *reseeding* [44], to maintain their usefulness;

- how the domain of the organizational memory can co–evolve with its content through a process called *incremental formalization* [99]; and
- how groups must come to a consensus on their filing patterns [8].

At present, Memento provides no direct support for the reorganization of an organizational memory stored therein. Memento does, however, provide several mechanisms that support the evolution of its information spaces and their domains that are described above. Finally, Memento does not address any social issues, such as group filing schemes. Additional support for organizational memories could be added as a software layer above the Memento infrastructure.

5.2 Human–Computer Interaction

Section 1.3 described the importance of the user interface with regard to the construction of assistants. The main points of that section are reviewed here.

The user interface serves as the conduit of meaning between the user and the assistant. Meaning is conveyed through the semantic level of the user interface via models that have representations both inside an assistant and in the mind of its user. Fischer and Lemke have used the term *human–problem domain communication* to describe this general idea [43, 41]. Memento supports human–problem domain communication in assistants through mechanisms which convey models and their domains via the semantic level of the user interface. These mechanisms, which were described in Section 3.5, make Memento one of very few systems that attempts to address the semantic level of the user interface.

Assistants often have a significant user interface component that is specialized for its domain. Construction of high quality user interfaces requires iterative development and user testing with each cycle feeding into the next [52]. Iterative development exerts an evolutionary force on both an assistant's domain and its user interface. Memento provides mechanisms to address the evolution of domains and their associated information spaces, described in Section 3.6.3. Additionally, Memento allows a clean separation of concerns in the construction of an assistant's user interface. Specifically, an assistant's semantic constraints are embodied in a *domain representation*, described in Section 4.1. An assistant's user interface component can rely on the domain representation for all semantic checking and manipulation and, therefore, need not duplicate any semantic constraint checking code. Presumably, a simpler user interface component would be more amenable to evolutionary change.

5.3 Software Engineering

In this section Memento's relation to several software engineering concepts is discussed. The Memento infrastructure is an object–oriented framework for assistants. Memento facilitates the reuse and evolution of an assistant's code. Finally, a designer may use the process of object–oriented analysis and design (OOAD) to create an assistant's domain representation.

5.3.1 Memento is a Framework

The Memento infrastructure is an object–oriented framework. An *object–oriented framework* (or just *framework*) is an embodiment of an abstract design that solves some general problem. Frameworks are built from an integrated collection of object–oriented classes that cooperate in the problem's solution. Programmers can customize and adapt a framework for particular needs through object–oriented inheritance by overriding the default behavior of some or all of the framework's classes. Each framework is intended to be reused in a variety of situations. A given framework can be specialized to solve a more specific problem. To solve more complex problems, frameworks for different problems can be combined like software integrated circuits [26].

Historically, the first software systems called frameworks were *application frameworks*. *Application frameworks* try to solve the general problem of building applications on a particular platform. Such a framework embodies the hard part of building an application. Application frameworks hide the the operating system issues like memory and resource management, low–level graphical primitives, and event loops. They assist with the details that all applications must handle: interaction with the clipboard, printing, window management, dialog boxes, and screen repainting. Application frameworks also usually contain a class library of user interface components. Ideally, an application framework embodies a good user interface standard that allows applications to inherit many of the properties of that standard. MacApp is a successful and well–known application framework for building Macintosh applications [95]. With most application frameworks, developers start from the "null application," which does everything an application must do in a completely generic way, and then they customize the application for their own needs.

Frameworks need not be limited, however, to the domain of building applications. In recent years, there has been an explosion in framework development. There are frameworks for a large variety of problems: financial systems, electronic circuit layout, graphical object editing, compilers, hypertext applications, operating systems, and so on. Each framework represents a substantial intellectual contribution and many are of academic interest. By now, enough frameworks been constructed to allow meaningful categorization and comparison [64].

Memento is a contribution to the study of frameworks in that it attempts to solve the novel problem of assisting with construction of the semantic level of user interfaces by facilitating collaboration over semantically meaningful information. Memento blends well with existing application frameworks which focus primarily on the syntactic and lexical levels of the user interface. Chapter 4 demonstrated how Memento could be combined with an application framework to form a complete assistant application.

5.3.2 Software Reuse

Software reuse is of paramount concern to software designers. Code that can be reused does not have to be written—a huge savings in effort. Software reuse is not completely free, however, as it does not happen by accident. The intent of reuse must be deliberately factored into the design and coding of the target software component [65].

Memento facilitates code reuse at three levels. First, the Memento infrastructure itself is intended to be used in a wide variety of assistants. Chapter 4 demonstrated Memento–based assistants in a variety of assistant categories. Next, when Memento is customized for a particular domain, the resulting portable *domain representation* can be reused in any context where the domain is applicable. For example, the same domain representation may be used by a collaborative assistant and any number of information agents that manipulate the same information space as the assistant. Customization of Memento for specific domains was discussed in Section 4.1. Finally, a suite of related assistants will likely be developed for complex domains. Assistants in the suite will likely share metamodels and portions of their respective domain representations. In such a situation, the assistants could possibly share reusable domain–specific user interface components, as well. Reuse of such high–level components is uncommon. Due to the limited scope of this dissertation, this third potential reuse scenario has yet to be demonstrated. Such a scenario might only arise after several years of usage of the Memento infrastructure.

5.3.3 Software Evolution

Change is an inevitable part of any successful computer system [13]. Section 1.3 identified the user interface component as an important source of change in assistants. Support for an assistant's evolution is an important goal and contribution of Memento.

Memento provides two mechanisms to support assistant evolution. First, the majority of an assistant's domain–specific customization is declarative and can be easily modified. The remainder of the *domain representation*, discussed in Section 4.1, consists mostly of straightforwardly implemented (and modified) convenience access and modification methods. Next, through the process of repository unification, described in Section 2.4.3 and Section 3.4, meaningful manipulation of models may be possible from outdated assistants. Several generations of assistants may gracefully manipulate the same information space. This allows new assistant versions to be phased–in slowly rather than requiring a coordinated upgrade to all assistant instances.

5.3.4 Object–Oriented Analysis and Design

Object–oriented analysis and design (OOAD) is a formalized process of software construction that facilitates the characterization of domain–specific systems. Several methodologies have been proposed for this process [10, 7]. The result of the object–oriented design is usually a set of diagrams that capture the essence of the software system to be built. It is often a straightforward process to translate the design diagrams into Memento metamodels, described in Section 3.2. Metamodels are used by Memento to embody the semantic constraints of a domain. By following this process, a software designer can quickly create a collaborative, domain representation for a given domain. The *domain representation* construction process was discussed in Section 4.1. Note that assistant developers are not required to use OOAD methodologies to create an assistant's metamodels. Algorithms are available to convert from other requirements formalisms to that of conceptual graph schemata [28].

5.4 Knowledge Representation

Knowledge representation is a broad area with many diverse contributions. A single, widely–accepted knowledge representation scheme was chosen as the basis for Memento's second layer. Sowa's conceptual graphs [103] have served Memento's semantic needs through a simple and extensible model that is amenable to simple graph pattern matching. Through the separation of Memento's first and second layers, Memento could be adapted for use with other knowledge representation schemes. A likely candidate is the well–known graph–based knowledge representation system KL-ONE [11].

Rather than survey the extensive area of knowledge representation in this section, attention is given to three systems that use knowledge representation schemes to achieve applications that augment the human intellect—systems that meet the definition of *assistants*. The three systems are: Fischer's Information Management System (IMS) developed at the University of Stuttgart; SHADE, from the Stanford Knowledge System Laboratory, mentioned earlier; and Graves' WEAVE System, developed at the University of Michigan.

The Information Management System (IMS), described in [40], is a programming environment that is based on an object–oriented LISP derivative called OBJTALK. Besides being used as the programming language for the program being developed, OBJTALK represents information about the greater context of the program including its problem domain, the design process, relevant information artifacts, and the IMS programming environment. While the IMS system has not had a big impact on later systems, the discussion in [40] of symbiotic, knowledge–based computer support systems has influenced the author's philosophy of assistants and their support by Memento.

SHADE [79], mentioned earlier, is an infrastructure for the sharing of engineering design information among computer–aided engineering systems. SHADE uses the LISP–like Knowledge Interchange Format (KIF) as its knowledge representation scheme. Like Memento, SHADE uses meaningful information artifacts as the basis for collaboration among cooperating applications. Unlike Memento, SHADE does not assume a centralized information space containing all knowledge, but rather views cooperating components as producers and consumers of different kinds of knowledge. A facilitation agent in SHADE, called a matchmaker, attempts to match consumers and producers. Apparently, one of the challenges in this approach is characterizing the kinds of knowledge a producer can supply. SHADE is part of a broader suite of tools as part of the ARPA Knowledge Sharing Initiative. Because of Memento's more modest goals, it is architecturally much simpler than SHADE. Memento, by contrast, assumes a shared, graph–structured information space over which various semantic–based graph patterns are applied.

Graves' WEAVE system [53, 54] starts with a formal knowledge specification and generates a functional programming language API to access and manipulate a domain–specific knowledge base. Memento's domain representation can also be thought of as a domain–specific API. WEAVE's specification language, WEB, extends constructive type theory to construct, access, and manipulate graph structures. The data model consists of unattributed nodes connected by binary relations. Among WEAVE's contributions is an efficient graph query algorithm over these graphs. Like Memento, WEAVE allows for overlapping graph schemata that provide views on a complex information space. WEAVE does not address collaboration as it assumes single application access. WEAVE also does not address the issue of evolution of either the knowledge base or the programs to manipulate it. In contrast, Memento supports collaboration and the evolution of both assistants and their associated information spaces.

5.5 Summary of Contributions

Memento is an infrastructure for supporting the construction and evolution of *assistant applications*, or *assistants*. Each *assistant* embodies an effective understanding of some information domain or problem domain. The assistant uses this understanding to aid a user or user community in the manipulation, transmission, and storage of, and user interaction with information artifacts from the assistant's domain, called *models*. When these services are combined, the result is artifact–based collaboration over an assistant's models. Memento attempts to preserve a model's meaning throughout these processes by enforcement of the domain's membership rules called *semantic constraints*. Memento represents the semantic constraints of a domain as data and is therefore *domain–independent*.

Memento improves the state of the art of building assistants in the following ways.

- Memento is an infrastructure that facilitates real–time, artifact–based collaboration over models from a variety of domains. This is made possible by the combination of Memento's two layers: the lower layer providing real–time artifact–based collaboration and the second layer representing models.

- Memento is a portable, reusable C++ framework that can be used in the construction of a wide variety of assistants, including information agents. Memento was used in the construction of several assistants described in Chapter 4.

- Memento abstracts the problem of meaning maintenance over models through the enforcement of a domain's semantic constraints. Chapter 3 described how semantic constraints are embodied in a set of metamodels that participate in Memento's transaction mechanism to preserve the meaning of their associated models.

- A domain's semantic constraints are represented in Memento as data. Memento is therefore *domain–independent*. Metamodels and the MType system are represented as MObjects.

- Memento is customized for an assistant's domain by creating a portable *domain representation*. A domain representation is mostly declarative and is easily modified. Creation and modification of an assistant's domain representation was described in Section 4.1.

- Memento provides mechanisms to facilitate evolution of an assistant's domain and associated information spaces. Section 3.6.3 described these mechanisms.

- Memento provides mechanisms for the communication of models and semantic constraints via the semantic level of the user interface. These mechanisms, described in Section 3.5, facilitate *human–problem domain communication* [41, 43] in Memento–based assistants.

Memento is the first infrastructure to systematically address the needs of assistant applications.

5.6 Conclusions and Future Work

Memento is a unique synergism of several key ideas: artifact–based collaboration, use of a knowledge representation scheme, emphasis on the semantic level of the user interface, and evolution of both an application and its information space. Many systems address only a subset of these issues. It is clear from this work, however, that all four issues must be dealt with for a truly successful approach in addressing the needs of assistant applications. The construction of assistants draws from a diversity of disciplines. As such, Memento is a clear demonstration that attention to related disciplines can positively impact the results produced in one.

Memento's two–level data model with its representation of meta–information as data has proven a successful approach for the coherent expression of a variety of domains. The MObject with its associated MType system are the lower level of the data model. Conceptual graphs built from MObjects comprise the upper level. Both levels allow for a general representation scheme geared towards different purposes though both are constrained by a dynamic type system. Typed collections of name–value attribute pairs, like Memento's MObject, are very common in systems for domain–independent representation. Also common is the construction of graph structures from the collections. Few systems, however, have a strong notion of a type system over such graphs.

The third success of Memento is the full utilization of a shared information space as a context for collaboration. Through real–time updates users of Memento–based assistants can immediately see the changes to the space made by others. The information space is persistently stored, so that a community can grow and shape the evolving information resource over time. Memento also facilitates the effective use of information agents in such a shared information space. They become full collaborators—effectively members of the user community. It is through such a shared community—containing both users and intelligent assistants—that the goal of intellectual synergism is realized.

Memento has greater potential than has been demonstrated in this dissertation. Future work on Memento will attempt to realize this potential in several areas. First, Memento might transparently support cross–domain interoperability through the simultaneous enforcement of related metamodels in a single information space shared by multiple assistants. Cross–domain interoperability is considered an important goal of computer–supported cooperative work [37]. This scenario was described in Section 3.6.2. Next, Memento may facilitate the reuse of high–level user interface components when shared between suites of related assistants. This type of potential software reuse was discussed in Section 5.3.2. Finally, Memento may facilitate the construction of entirely new types of assistants. By easing the burden of building assistants, developers will be able to concentrate on more complex assistants. Collaborative information agents is a ripe area for further exploration.

Bibliography

[1] Ronald M. Baecker, editor. *Readings in Groupware and Computer–Supported Cooperative Work*. Morgan Kaufmann Publishers, Inc., San Mateo, California, 1993.

[2] Donald Baker, Jerry Fowler, Vram Kouramajian, Ross Dargahi, Gwyn Guidy, and Tom Lytle. Memento: A collaborative metaphor–independent architecture. Technical Report TR94–230, Rice University Computer Science Department, 6100 Main St., Houston, TX 77005–1892, February 1994.

[3] Donald G. Baker, Jerry Fowler, and G. Anthony Gorry. The Memento collaboration infrastructure. Technical Report CTTL–96–1, Center for Technology in Teaching and Learning, Rice University, 6100 Main St., Houston, TX 77005–1892, 1996.

[4] Liam J. Bannon and Kjeld Schmidt. CSCW: Four characters in search of a context. In J. M. Bowers and S. D. Benford, editors, *Studies in Computer Supported Cooperative Work: Theory, Practice, and Design. Proceedings of the First European Conference on Computer Supported Cooperative Work*, pages 3–16, North–Holland, 1991. Elsevier Science Publishers. Also published in [1].

[5] Douglas Barry. Charting the feature coverage of odbmss. *Object Magazine*, 6(12):29–42, February 1997.

[6] David Beard, Murugappan Palaniappan, Alan Humm, David Banks, Anil Nair, and Yen-Ping Shan. A visual calendar for scheduling group meetings. In *Proceedings of the ACM 1990 Conference on Computer Supported Cooperative Work (CSCW '90)*, pages 279–290, October 1990.

[7] Edward V. Berard. *Essays on Object–Oriented Software Engineering*, volume 1. Prentice Hall, Englewood Cliffs, New Jersey, 1993.

[8] Lucy M. Berlin, Robin Jeffries, Vicki L. O'Day, Andreas Paepcke, and Cathleen Wharton. Where did you put it? Issues in the design and use of a group memory. In *Human Factors in Computing Systems: INTERCHI '93 Conference Proceedings*, pages 23–30. ACM, 1993.

[9] Daniel G. Bobrow, Sanjay Mittal, and Mark J. Stefik. Expert systems: Perils and promise. *Communications of the ACM*, 29(9):880–894, September 1986.

[10] Grady Booch. *Object–Oriented Design with Applications*. Benjamin Cummings Publishing Company, Redwood City, California, 1991.

[11] R. J. Brachman and J. G. Schmolze. An overview of the KL–ONE knowledge representation system. *Cognitive Science*, 9(2):171–185, 1985.

[12] Brøderbund Software, Inc., 500 Redwood Blvd., Novato, California 94948–6121. *Myst*, Macintosh edition, 1993.

[13] Frederick P. Brooks, Jr. No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, April 1987.

[14] L. Brothers, V. Sembugamoorthy, and M. Muller. ICICLE: Groupware for code inspection. In *Proceedings of the ACM 1990 Conference on Computer Supported Cooperative Work (CSCW '90)*, pages 169–181, October 1990.

[15] Eric W. Brown, James P. Callan, and W. Bruce Croft. Fast incremental indexing for full–text information retrieval. In *Proceedings of the 20th International Conference on Very Large Databases (VLDB)*. ACM, 1994.

[16] Vannevar Bush. As we may think. *The Atlantic Monthly*, 176(1):101–108, July 1945.

[17] Stuart K. Card, George G. Robertson, and Jock D. Mackinlay. The information visualizer, an information workspace. In *Human Factors in Computer Systems: CHI '91 Conference Proceedings*, pages 181–188, New Orleans, LA, April 1991.

[18] Michael J. Carey, Laura M. Haas, Peter M. Schwarz, Manish Arya, William F. Cody, Ronald Fagin, Myron Flickner, Allen W. Luniewski, Wayne Niblack, Dragutin Petkovic, John Thomas, John H. Williams, and Edward L. Wimmers. Towards heterogeneous multimedia information systems: The garlic approach.

In *Proceedings of the Fifth International Workshop on Research Issues in Data Engineering (RIDE): Distributed Object Management*, March 1995.

[19] John M. Carroll, Robert L. Mack, and Wendy A. Kellogg. Interface metaphors and user interface design. In Martin Helander, editor, *Handbook of Human–Computer Interaction*, chapter 3, pages 67–85. Elsevier Science Publishers, North Holland, 1988.

[20] John M. Carroll and Judith Reitman Olson. Mental models in human–computer interaction. In M. Helander, editor, *Handbook of Human–Computer Interaction*, chapter 2, pages 45–65. Elsevier Science Publishers, North Holland, 1988.

[21] R. G. G. Cattell. *Object Data Management*. Addison–Wesley Publishing Company, Reading, Massachusetts, 1994.

[22] Peter Pin-Shan Chen. The entity-relationship model—toward a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–36, 1976.

[23] Jeff Conklin and Michael L. Begeman. gIBIS: A hypertext tool for team design deliberation. In *Hypertext '87 Papers*, pages 247–251. ACM, November 1987.

[24] Jeff Conklin and Michael L. Begeman. gIBIS: A hypertext tool for exploratory policy discussion. *ACM Transactions on Office Information Systems*, 6(4), October 1988. Also appears in the 1988 Conference on Computer Supported Cooperative Work (CSCW '88).

[25] Nancy Jaworski Cooke. A methodology for capturing the illusive mental model. In *Proceedings of USICON '88, 3rd Annual User–System Interface Conference*, pages 167–175, 1988.

[26] Brad J. Cox and Andrew J. Novobilski. *Object Oriented Programming: an Evolutionary Approach*. Addison Wesley, Reading, Massachusetts, 1991.

[27] Pavel Curtis and David A. Nichols. MUDs grow up: Social virtual reality in the real world. In *Proceedings of the Third International Conference on Cyberspace*, May 1993.

[28] Harry S. Delugach. Specifying multiple–viewed software requirements with conceptual graphs. *Journal of Systems Software*, 19(3):207–224, November 1992.

[29] A. R. Dennis, F. G. Joey, L. M. Jessup, J. F. Nunamaker, and D. R. Vogel. Information technology to support electronic meetings. *MIS Quarterly*, 12(4):591–619, December 1988.

[30] Prasun Dewan. Tools for implementing multiuser user interfaces. In Len Bass and Prasun Dewan, editors, *User Interface Software*, chapter 8, pages 149–174. John Wiley and Sons, Ltd., Chichester, West Sussex, England, 1993.

[31] Prasun Dewan and Rajiv Choudhary. A high–level and flexible framework for implementing multiuser user interfaces. *ACM Transactions on Information Systems*, 10(4):345–380, October 1992.

[32] Digital Equipment Corporation. *Linkworks Version 3.0 User's Guide*, 1995.

[33] Paul Dourish. Consistency guarantees: Exploiting application semantics for consistency management in a collaboration toolkit. In *Proceedings of the ACM 1996 Conference on Computer Supported Cooperative Work (CSCW '96)*, pages 268–277, November 1996.

[34] Ernest A. Edmonds, Linda Candy, Rachel Jones, and Bassel Soufi. Support for collaborative design: Agents and emergence. *Communications of the ACM*, 37(7):41–47, July 1994.

[35] C. A. Ellis, S. J. Gibbs, and G. L. Rein. Groupware: Some issues and experiences. *Communications of the ACM*, 34(1):38–58, January 1991.

[36] D. C. Engelbart and W. K. English. A research center for augmenting the human intellect. In *AFIPS Conference Proceedings, 1968 Fall Joint Computer Conference*, pages 395–410, San Francisco, CA, December 1968.

[37] Douglas C. Engelbart. Knowledge–domain interoperability and an open hyperdocument system. In *Proceedings of the ACM 1990 Conference on Computer Supported Cooperative Work (CSCW '90)*, pages 143–156, October 1990.

[38] George Ernst and Allen Newell. *GPS: A Case Study in Generality and Problem Solving*. Academic Press, New York, 1969.

[39] Michael Feldman and George T. Rogers. Towards the design and development of style–independent interactive systems. In *Human Factors In Computer Systems: CHI '82 Conference Proceedings*, pages 111–116. ACM, 1982.

[40] Gerhard Fischer. Symbiotic, knowledge–based computer support systems. *Automatica*, 19(6):627–637, June 1983.

[41] Gerhard Fischer. Human–computer interaction software: Lessons learned, challenges ahead. *IEEE Software*, 6(1):44–52, January 1989.

[42] Gerhard Fischer, Andreas Girgensohn, Kumiyo Nakakoji, and David Redmiles. Supporting software designers with integrated domain–oriented design environments. *IEEE Transactions on Software Engineering*, 18(6):511–522, June 1992.

[43] Gerhard Fischer and Andreas C. Lemke. Construction kits and design environments: Steps toward human problem–domain communication. *Human–Computer Interaction*, 3(3):179–222, 1987–1988.

[44] Gerhard Fischer, Ray McCall, Jonathan Ostwald, Brent Reeves, and Frank Shipman. Seeding, evolutionary growth and reseeding: Supporting the incremental development of design environments. In *Human Factors in Computing Systems: CHI '94 Conference Proceedings*, pages 292–298. ACM, April 1994.

[45] R. Fish, R. Kraut, M. Leland, and M. Cohen. Quilt: A collaborative tool for cooperative writing. In *Proceedings of the Conference on Office Information Systems*, pages 30–37, New York, 1988. ACM.

[46] Geraldine Fitzpatrick, Simon Kaplan, and Tim Mansfield. Physical spaces, virtual places and social worlds: A study of work in the virtual. In *Proceedings of the ACM 1996 Conference on Computer Supported Cooperative Work (CSCW '96)*, pages 334–343, November 1996.

[47] Geraldine Fitzpatrick, William Tolone, and Simon M. Kaplan. Work, locales and distributed social worlds. In H. Marmolin, Y. Sundblad, and K. Schmidt, editors, *Proceedings of the Fourth European Conference on Computer–Supported Cooperative Work*, pages 1–16. Kluwer Academic Publishers, September 1995.

[48] James Foley, Christina Gibbs, Won Chul Kim, and Srdjan Kovacevic. A knowledge–based user interface management system. In *Human Factors in Computer Systems: CHI '88 Conference Proceedings*, pages 67–72, May 1988.

[49] Jerry Fowler. *The Virtual Object Model for Distributed Hypertext*. PhD thesis, Rice University Computer Science Department, 6100 Main St., Houston, TX 77005–1892, 1994.

[50] Jerry Fowler, Donald G. Baker, Vram Kouramajian, Hillary Gilson, Ross Dargahi, Kevin Brook Long, Cynthia Petermann, and G. Anthony Gorry. Experience with the virtual notebook system: Abstraction in hypertext. In *Proceedings of the ACM 1994 Conference on Computer Supported Cooperative Work (CSCW '94)*, pages 133–143, October 1994.

[51] Adele Goldberg and David Robson. *Smalltalk 80: The Language and its Implementation.* Addison–Wesley, Reading, MA, 1983.

[52] John D. Gould and Clayton Lewis. Designing for usability—key principles and what designers think. In *Human Factors in Computing Systems: CHI '83 Conference Proceedings*, pages 50–53. ACM, November 1983.

[53] Mark Graves. *Theories and Tools for Designing Application–Specific Knowledge Base Data Models.* PhD thesis, Department of Electrical Engineering and Computer Science, University of Michigan, 1993.

[54] Mark Graves. Application of knowledge base design techniques to genetic markers. In Niki Pissinou, Ave Silberschatz, E. K. Park, and Kia Makki, editors, *Proceedings of the 1995 ACM CIKM Fourth International Conference on Information and Knowledge Management*, pages 348–354, November–December 1995.

[55] Kaj Grønbaek, Jens A. Hem, Ole L. Madsen, and Lennert Sloth. Designing dexter–based cooperative hypermedia systems. In *Hypertext '93 Proceedings: The Fifth ACM Conference on Hypertext*, pages 25–38, November 1993.

[56] Anja Haake and Jörg M. Haake. Take CoVer: Exploiting version support in cooperative systems. In *Human Factors in Computing Systems: INTERCHI '93 Conference Proceedings*, pages 406–413. ACM, 1993.

[57] Frank Halasz, Thomas P. Moran, and Randall H. Trigg. Notecards in a nutshell. In *Proceedings, ACM CHI+GI '87 Conference Proceedings Human Factors in Computing Systems and Graphics Interface*, pages 45–52, 1987.

[58] David Harel. On visual formalisms. *Communications of the ACM*, 31(5):514–530, May 1988.

[59] Peggy M. Irish and Randall H. Trigg. Supporting collaboration in hypermedia: Issues and experiences. In Edward Barrett, editor, *The Society of Text*, pages 90–106. MIT Press, Cambridge, Massachusetts, 1989.

[60] K. Jeffay, J. K. Lin, J. Menges, F. D. Smith, and J. B. Smith. Architecture of the artifact–based collaboration system matrix. In *Proceedings of the ACM 1992 Conference on Computer Supported Cooperative Work (CSCW '92)*, pages 195–202, October–November 1992.

[61] Jeff Johnson. Selectors: Going beyond user–interface widgets. In *Human Factors in Computer Systems: CHI '92 Conference Proceedings*, pages 273–280, 1992.

[62] Philip Johnson. Supporting exploratory CSCW with the EGRET framework. In *Proceedings of the ACM 1992 Conference on Computer Supported Cooperative Work (CSCW '92)*, pages 298–305. ACM, October–November 1992.

[63] Philip Johnson. EGRET: A framework for advanced CSCW applications. *Software Engineering Notes*, 21(5):70–71, September 1996.

[64] Ralph E. Johnson and Brian Foote. Designing reusable classes. *Journal of Object Oriented Programming*, 1(1), June/July 1988.

[65] Ralph E. Johnson and Vincent F. Russo. Reusing object–oriented designs. Technical Report UIUCDSC 91–1696, University of Illinois, Department of Computer Science, May 1991.

[66] Wolfgang Klas, Karl Aberer, and Erich Neuhold. Object–oriented modelling for hypermedia systems using the VODAK model language. In A. Dogac, T. Ozsu, A. Biliris, and T. Sellis, editors, *Advances in Object–Oriented Database Systems*. Springer–Verlag Berlin, Heidelberg, Germany, 1994.

[67] Michael J. Knister and Atul Prakash. DistEdit: A distributed toolkit for supporting multiple group editors. In *Proceedings of the ACM 1990 Conference on Computer Supported Cooperative Work (CSCW '90)*, pages 343–355, October 1990.

[68] Vram Kouramajian, Ross Dargahi, Donald Baker, and Jerry Fowler. Consortium: A framework for transactions in collaborative environments. In *Proceed-*

ings of the 1995 ACM CIKM: Fourth International Conference on Information and Knowledge Management, pages 260–265, December 1995.

[69] Richard T. Kouzes, James D. Myers, and William A. Wulf. Collaboratories: Doing science on the Internet. *IEEE Computer*, 29(8):40–46, August 1996.

[70] K. L. Kraemer and J. L. King. Computer–based systems for cooperative work and group decision making. *ACM Computing Surveys*, 20(2):115–146, June 1988.

[71] Glenn E. Krasner and Stephen T. Pope. A cookbook for using the model–view–controller user interface paradigm in Smalltalk–80. *Journal of Object Oriented Programming*, 1(3):26–49, August–September 1988.

[72] K.-Y. Lai, T. W. Malone, and K.-C. Yu. Object Lens: A 'spreadsheet' for cooperative work. *ACM Transactions on Office Information Systems*, 6(4):332–353, 1988.

[73] George P. Landow. Hypertext and collaborative work: The example of intermedia. In J. Galegher, R. Kraut, and C. Egido, editors, *Intellectual Teamwork: Social and Technological Foundations of Cooperative Work*, pages 407–415. Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1990. Excerpts published in [1].

[74] Jintae Lee. SIBYL: A tool for managing group design rationale. In *Proceedings of the ACM 1990 Conference on Computer Supported Cooperative Work (CSCW '90)*, pages 79–92, October 1990.

[75] Lotus Development Corporation. *Lotus Notes User's Guide*, 1994.

[76] Thomas W. Malone, Kum-Yew Lai, and Christopher Fry. Experiments with Oval: A radically tailorable tool for cooperative work. In *Proceedings of the ACM 1992 Conference on Computer Supported Cooperative Work (CSCW '92)*, pages 289–297, October–November 1992.

[77] Thomas W. Malone, Kum-Yew Lai, and Christopher Fry. Experiments with Oval: A radically tailorable tool for cooperative work. *ACM Transactions on Information Systems*, 13(2):177–205, April 1995.

[78] Aaron Marcus. Human communications issues in advanced UIs. *Communications of the ACM*, 36(4):101–109, April 1993.

[79] James G. McGuire, Daniel R. Kuokka, Jay C. Weber, Jay M. Tenenbaum, Thomas R. Gruber, and Gregory R. Olsen. SHADE: Technology for knowledge–based collaborative engineering. *Journal of Concurrent Engineering: Research and Applications (CERA)*, 1(2), 1993.

[80] Norman Meyrowitz. Intermedia: The architecture and construction of an object–oriented hypermedia system and applications framework. In *OOPSLA '86 Conference Proceedings*, pages 186–201. ACM, November 1986. Also catalogued as SIGPLAN Notices, 21(11).

[81] Microsoft Corporation, Redmond, Washington. *The Microsoft Foundation Class Library*, 1991.

[82] Guy W. Mineau, Bernard Moulin, and John F. Sowa, editors. *Conceptual Graphs for Knowledge Representation; First International Conference on Conceptual Structures (ICCS '93)*, Berlin, August 1993. Springer–Verlag. Lecture Notes in Artificial Intelligence #699.

[83] David J. Musliner, James W. Dolter, and Kang G. Shin. BIBDB: A bibliographic database for collaboration. In *Proceedings of the ACM 1992 Conference on Computer Supported Cooperative Work (CSCW '92)*, pages 386–393, October–November 1992.

[84] Bonnie A. Nardi and Craig L. Zarmer. Beyond models and metaphors: Visual formalisms in user interface design. In *Proceedings of the 24th Annual Hawaii International Conference on Systems Science*, volume 2, pages 478–493, Koloa, HI, January 1991.

[85] Donald A. Norman. Some observations on mental models. In Albert L. Stevens and Dedre Gentner, editors, *Mental Models*, chapter 1, pages 7–14. Lawrence Erlbaum Associates, Hillsdale, N. J., 1983.

[86] Object Management Group, Framingham, MA. *The Common Object Request Broker: Architecture and Specification*, 1.1 edition, 1992.

[87] OpenDoc Design Team. *The OpenDoc Technical Summary.* Apple Computer Inc., 1994.

[88] David A. Patterson. Microprocessors in 2020. *Scientific American*, 273(3):62–67, September 1995.

[89] John F. Patterson, Ralph Hill, and Steven L. Rohall. Rendezvous: An architecture for synchronous multi–user applications. In *Proceedings of the ACM 1990 Conference on Computer Supported Cooperative Work (CSCW '90)*, pages 317–328, October 1990.

[90] Heather D. Pfeiffer and Tomothy E. Nagle, editors. *Conceptual Structures: Theory and Implementation; 7th Annual Workshop*, Berlin, July 1992. Springer–Verlag. Lecture Notes in Artificial Intelligence #754.

[91] Atul Prakash and Hyong Sop Shim. DistView: Support for building efficient collaborative applications using replicated objects. In *Proceedings of the ACM 1994 Conference on Computer Supported Cooperative Work (CSCW '94)*, pages 153–164, October 1994.

[92] Brent Reeves and Frank Shipman. Supporting communication between designers with artifact–centered evolving information spaces. In *Proceedings of the ACM 1992 Conference on Computer Supported Cooperative Work (CSCW '92)*, pages 394–401, Toronto, Canada, November 1992.

[93] George G. Robertson, Stuart K. Card, and Jock D. Mackinlay. Information visualization using 3D interactive animation. *Communications of the ACM*, 36(4):57–71, April 1993.

[94] Mark Roseman and Saul Greenberg. GroupKit: A groupware toolkit for building real–time conferencing applications. In *Proceedings of the ACM 1992 Conference on Computer Supported Cooperative Work (CSCW '92)*, pages 43–50, October–November 1992.

[95] Kurt J. Schmucker. Macapp: An application framework. *BYTE*, 11(8):189–193, August 1986.

[96] Eve M. Schooler. Conferencing and collaborative computing. *Multimedia Systems*, 4(5):210–225, October 1996.

[97] Christian Schuckmann, Lutz Kirchner, Jan Schümmer, and Jörg M. Haake. Designing object–oriented synchronous groupware with COAST. In *Proceedings of the ACM 1996 Conference on Computer Supported Cooperative Work (CSCW '96)*, pages 30–38, November 1996.

[98] Douglas E. Shackelford, John B. Smith, and F. Donelson Smith. The architecture and implementation of a distributed hypermedia storage system. In *Hypertext '93 Proceedings: The Fifth ACM Conference on Hypertext*, pages 1–13, November 1993.

[99] Frank M. Shipman, III and Raymond McCall. Supporting knowledge–base evolution with incremental formalization. In *Human Factors in Computing Systems: CHI '94 Conference Proceedings*, pages 285–291. ACM, April 1994.

[100] Ben Shneiderman. *Designing the User Interface: Strategies for Effective Human–Computer Interaction*. Addison–Wesley Publishing Company, Reading, Massachusetts, 1987.

[101] Edward Shortliffe. *Computer Based Medical Consultations: MYCIN*. Elsevier Science Publishers, New York, 1976.

[102] John B. Smith and F. Donelson Smith. Abc: A hypermedia system for artifact–based collaboration. In *Hypertext '91 Proceedings*, pages 179–192. ACM, December 1991.

[103] John F. Sowa. *Conceptual Structures: Information Processing in Mind and Machine*. Addison–Wesley Publishing Company, Reading, Massachusetts, 1984.

[104] Mark Stefik, Gregg Foster, Daniel G. Bobrow, Kenneth Kahn, Stan Lanning, and Lucy Suchman. Beyond the chalkboard: Computer support for collaboration and problem solving in meetings. *Communications of the ACM*, 30(1):32–46, January 1987.

[105] Norbert Streitz, Jörg Haake, Jörg Hannemann, Andreas Lemke, Wolfgang Schuler, Helge Schütt, and Manfred Thüring. SEPIA: A cooperative hypermedia authoring environment. In *Proceedings of the Fourth ACM Conference on Hypertext (ECHT '92)*, pages 11–22, November–December 1992.

[106] T. J. Teorey, D. Yang, and J. P. Fry. A logical design methodology for relational databases using the extended entity–relationship model. *ACM Computing Surveys*, 18(2):197–222, June 1986.

[107] Loren G. Terveen, Peter G. Selfridge, and M. David Long. From 'folklore' to 'living design memory'. In *Human Factors in Computing Systems: INTERCHI '93 Conference Proceedings*, pages 15–22. ACM, 1993.

[108] William J. Tolone, Simon M. Kaplan, and Geraldine Fitzpatrick. Specifying dynamic support for collaborative work in WORLDS. In *Conference on Organizational Computing Systems*, pages 55–65. ACM, August 1995.

[109] Dadong Wan and Philip M. Johnson. Computer supported collaborative learning using CLARE: the approach and the experimental findings. In *Proceedings of the ACM 1994 Conference on Computer Supported Cooperative Work (CSCW '94)*, pages 187–198, October 1994.

[110] K. C. Burgess Yakemovic and E. Jeffery Conklin. Report on a development project use of an issue–based information system. In *Proceedings of the ACM 1990 Conference on Computer Supported Cooperative Work (CSCW '90)*, pages 105–118, October 1990. Also published in [1].

[111] Craig L. Zarmer and Chee Chew. Frameworks for interactive, extensible, information–intensive applications. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST '92)*, pages 33–41, 1992.