Hadez, a Framework for the Specification and Verification of Hypermedia Applications

by

Daniel Morales Germán

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Doctor in Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2000

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

In recent years, several methodologies for the development of hypermedia applications have been proposed. These methodologies are, primarily, guidelines to be followed during the design process. They also indicate what deliverables should be created at each of their stages. These products are usually informally specified—in the sense that they do not have formal syntax nor formally defined semantics—and they are not required to pass validity tests.

Hadez formally specifies the design of a hypermedia application, supports the verification of properties of the specification, and promotes the reuse of design.

Hadez is an object-oriented specification language with formal syntax and semantics. Hadez is based on the formal specification languages Z and Z++, with extensions unique to hypermedia. It uses set theory and first order predicate logic. It divides the specification of a hypermedia application into three main parts: its conceptual schema, which describes the domain-specific data and its relationships; its structural schema, which describes how this data is combined and gathered into more complex entities, called composites; and the perspective schema, which uses Abstract Design Perspectives (artifacts unique to Hadez) to indicate how these composites are mapped to hyperpages, and how the user interacts with them.

Hadez provides a formal framework in which properties of a specification can be specified and answered.

The specification of an application should not constrain its implementation and, therefore, it is independent of the platform in which the application is to be presented. As a consequence, the same design can be instantiated into different applications, each for a different hypermedia platform.

Hadez can be further extended with design patterns. Patterns enable reuse by capturing good solutions to well-known problems. Hadez characterizes patterns and makes their use readily available to the designer.

Furthermore, Hadez is process independent, and is intended to be used with any of the main hypermedia design methodologies: EROM, HDM, OOHDM or RMM.

A Mael,
Yo no lo sé de cierto...

Acknowledgements

As a child, I dreamed of becoming a scientist. I have been very fortunate that destiny gave me that opportunity. It is impossible in few paragraphs to acknowledge and thank everyone who has had an influence on me and who helped me achieved this degree. Some people, however, deserve to be mentioned.

I owe much to my parents, who provided me with a loving environment in which, as I child and a teenager, I could thrive.

I thank Don Cowan, my supervisor, who during my stay at UW has been like a second father to me. I thank him for his guidance and his support during these years.

I thank my committee members—Paulo Alencar, Joanne Atlee, Paul Beam and Franca Garzotto—for all their time and help in the evaluation and improvement of this manuscript.

Again, I thank Paulo Alencar, whose advice was invaluable during the development of this project.

My professors and teachers have a special place in my heart. They have been the true architects of my professional career. Three of them stand above all: H. C. Marcell incited my curiosity at an early stage and made me realize that my future lay in the sciences. My undergraduate professor, boss, mentor, friend and older brother, Noé Rodríguez, showed me that life is a rich experience that we live only once. Phil Kearns has been my model of what a professor should be; and he made me realize that I had what it takes to become a doctor.

This thesis is as much mine as it is of my lifetime friends: Clau, Letty, and Óscar. They always shared my dream of one day seeing this document materialize. Many thanks to my UW friends: Alex, Claudia, Francisco, Guta, Joy, Luis, Marcelino, Maureen and Torsten, who shared with me the joys and tears of graduate school. I thank, once again, Joy for helping me in the early stages of this document, and Claudia for her helpful comments on its final drafts.

I thank both IBM Canada and ICR/CITO for their financial support, and the Department of Computer Science at the University of Waterloo for giving me the opportunity of a lifetime.

Finally, I thank Donna Randall, whose love, patience, support, and encouragement help me go through the end.

And in this piece of yours... there is no need for you to go a-begging for aphorisms from philosophers, precepts from Holy Scripture, fables from poets, speeches from orators, or miracles from saints; but merely to take care that your style and diction run musically, pleasantly, and plainly, with clear, proper, and well-placed words, setting forth your purpose to the best of your power, and putting your ideas intelligibly, without confusion or obscurity.

Y, en esta vuestra escritura... no hay para qué andéis mendigando sentencias de filósofos, consejos de la Divina Escritura, fábulas de poetas, oraciones de retóricos, milagros de santos, sino procurar que a la llana, con palabras significantes, honestas y bien colocadas, salga vuestra oración y período sonoro y festivo; pintando, en todo lo que alcanzaréis y fuere posible, vuestra intención, dando a entender vuestros conceptos sin intricarlos y escurecerlos.

Miguel de Cervantes Saaveedra
Don Quijote, 1605

Contents

List of Figures

List of Tables

We lack guidelines and tools to design and create hypermedia applications which involve frequently changing information. Without such design guidelines and tools, the ever-growing network of interlinked applications [such as the World-Wide Web] is becoming increasingly spaghetti-like and hard to maintain.

Bieber and Isakowitz [BI95]

CHAPTER 1

Introduction

J.L. Borges wrote in *The Garden of the Forking Paths* (published in 1941) about stories that would be like a labyrinth: at each crossing point the reader could choose one of multiple options, the choice of the readers would determine their future reading experience while the writer's job would be to create diverse futures, which proliferate and fork. A writer would be like a labyrinth designer [Bor98].

Few years later, in 1945, Vannevar Bush dreamed of a machine that could store books, periodicals, personal correspondence, or any kind of printed records; then organize them and make them easily available to the user. The most remarkable feature of the *Memex*—as he chose to call it—was the ability to "tie

two items together... At any time, when one of these items is in view, the other can be instantly recalled merely by tapping a button below the corresponding code space." [Bus45]. Bush predicted that "wholly new forms of encyclopedias will appear, ready made with a mesh of associative trails running through them, ready to be dropped into the Memex and there amplified." Long before T. Nelson coined the term hypertext [Nel65], Borges and Bush imagined a system that would allow the reader to find its own way through information systems by deciding which link to follow, and, at the same time, they realized the complexities of authoring these systems.

The design of hypertext and hypermedia[1] applications is a difficult task, particularly the design of large, data-intensive, evolving systems. The task of the designers is not to interlink everything; instead, they should interconnect the parts of the system in a way that convey the overall meaning of the application in a natural way [GPS93]. This problem is compounded by the growing size of applications. Small applications could be handcrafted, but large applications require frameworks, tools, techniques, methods and metrics that guarantee their high quality; for large applications, it is necessary to adopt a more formal approach [GLR95].

1.1 Motivation

The exponential growth of the World-Wide Web has transformed the design and development of hypermedia systems into common tasks. The Web has become, by far, the most common hypermedia platform. Web applications range from simple personal home sites—composed of a few pages—to corporate web sites that are composed of millions of pages. The design and development of these hypermedia applications has been recognized as a difficult process, specially for large applications [BV97]. Many of these applications are created as a view of a database. The database is queried and the results are converted into hypermedia nodes that are crosslinked.

One of the most important attempts to cope with the complexity of the design of these applications is the adaptation of software engineering techniques

[1] In this dissertation we use both terms interchangeably.

to hypermedia design. One of these efforts is the creation of design methodologies (such as Isakowitz et al.'s Relational Management Methodology—RMM—[ISB95], Schwabe and Rossi's Object-Oriented Hypermedia Design Model—OOHDM—[SR95], Lange's Enhanced Object-Relationship Model—EORM—[Lan94], and Garzotto et al.'s Hypermedia Design Model—HDM—[GMP95a]).

These methodologies are guidelines to be followed during the design process. They also specify the characteristics of the deliverables, which are created at each of their stages. These products usually are not formally specified—in the sense that they do not have formal syntax nor formally defined semantics—and they are not required to pass validity tests.

It is important that such methodologies use well-defined formal descriptions for their deliverables in order to facilitate the creation of support tools and the verification of properties of the design (such as whether the specification is syntactically and semantically correct, is complete or is type consistent, for example). Formal methods have been used in the past to specify the time constraints of dynamic hypermedia applications ([CDOS96, SSGC98, PTdOM98]), and they have been used to specify specific characteristics of the hypermedia platform [dH97]. They have not been used to specify the deliverables of the hypermedia methodologies for discrete hypermedia applications. Discrete hypermedia applications are those which do not have dynamic or time-based components.

Specifications have different purposes [vHL89]:

- They serve as documentation.
- They serve as a mechanism for generating questions.
- They can be used as a contract between the designer and the customer.
- Good specifications facilitate implementation and maintenance.

Specifications can be informal and formal. Informal specifications—in particular those written in a natural language—present deficiencies that make them unsuitable for rigorous development of applications [Mey85]. Formal specifications, on the other hand, use a mathematical notation to describe, precisely and unambiguously, the properties which an information system must have without unduly constraining the way in which those properties are achieved [Spi92a].

Questions regarding the characteristics of the final application can be answered with confidence by analyzing the design instead of analyzing the final application; furthermore, formal specifications are unambiguous, contrary to diagrammatic or textual specifications.

In [DP94], Dospisil and Polgar described the main requirements of a specification language for hypermedia design: 1) it should be a cognitive model rather than a design or implementation model; 2) it should be possible to use simple mechanical methods to verify formal descriptions; 3) it should be possible for an inexperienced user to participate in the specification of the system; 4) the notation should not be closed; and, 5) it should be understandable even in very large systems.

A formal specification language "provides the means of precisely defining notions like consistency and completeness, and more relevantly, specification, implementation and correctness. It provides the means of proving that a specification is realizable, proving that a system has been implemented correctly, and proving properties of a system without necessarily running it to determine its behavior." [Win90]

A formal specification language (FSL) for hypermedia design would allow a developer to specify a hypermedia application unambiguously and to verify certain properties about it. Such a language will assist the designer to find errors in the design which otherwise can only be found during the implementation or testing of the application. Maintenance is also improved. In the first place, the specification serves as documentation, and second, the ability to verify the specification will allow the maintainer to realize the impact of changes in the design before actually implementing them. When following a formal method, the specification of a system is the most important part of its design and development [Hal90].

In the scope of this research, we restrict the definition of a hypertext application to a collection of nodes with text as its "basic component", with the addition of images and other static visual components. The nodes are referenced amongst themselves by links and anchors that are embedded into the text, or surrounding any of the other components (in this case they have a visual representation, such as images); and the links can be typed and bi-directional. This type of hypertext is known as discrete or static hypertext [Pra97]. Even though it does not include dynamic media (such as sound or video clips), this

type of hypertext is still sufficient for the majority of the hypertext applications in common hypertext architectures such as the World-Wide Web and interactive CD-ROMs, and less sophisticated platforms such as PDF, Windows HELP, and Emacs Info.

1.1.1 The Four Axes model

An application is typically composed of four different types of information:

- Content. The main goal of the application is to present this data, hence it is the core of the application.
- Structure. Depending on the objective of the application, the content must be organized in a manner that makes sense. This structure is usually hierarchical in nature. For instance, a museum is composed of virtual tours and each of the virtual tours is composed of paintings.
- Navigation. The structure of the application must be broken into hyperpages, and these pages are crosslinked. In the case of the museum, all the tours of the museum can be presented in a single, large page or they can be presented in multiple pages. The navigation determines how the structure of the application is broken into individual hyperpages.
- Presentation. Once an application is broken into individual hyperpages, each of these is composed of one or more data attributes. The presentation determines, first, which attributes are presented and which are hidden from the reader; and second, how these attributes are organized into the page, and third how they are typeset. For instance, assume that a hyperpage is composed of data a, b and c, it might be desirable to only show a and b; furthermore, a might be shown with large type and in red color, while b might be shown in normal type and black color. The presentation of a hyperpage might also involve interaction with the user, to determine the presentation of a particular hyperpage.

Changes in some of these axes can affect others, but in general, we can consider each of them—content, structure, navigation and presentation—as orthogonal to each other. We call this division the Four Axes model. The sep-

aration of the design and implementation in terms of these axes has several advantages:

- It is consistent with the principle of separation of concerns [Aks96].
- The content can be presented in multiple hypermedia applications. For example, the content for a hypermedia encyclopedia might be presented in a large version, and small version, a children's version, a CD-ROM, a web site, etc.
- The same content can be presented in many different structures, depending on the goal of the application. For example, it is possible to group articles in the encyclopedia by author, by theme, by year in which they were written, etc.
- Navigation, too, can take different forms. For example, a very simple type of navigation is the table of contents and index, typically found in Adobe's PDF files. Navigation is tightly coupled with the run-time platform in which the application is going to run. By separating the navigation from the content and structure we guarantee that the same application can be implemented for different run-time platforms.
- In terms of presentation, different readers might have different presentation requirements; for example: children seem to favor highly sophisticated presentation; personal digital assistants, on the other hand, require simplified typesetting, while visually impaired people require text-only applications and find it almost impossible to traverse graphics-oriented hypermedia applications. Similarly, different presentation might achieve different goals; for instance, an application that only shows titles can give a very fast overview, while an application that shows titles and paragraphs can give a complete perspective of the same information.

Many models of hypermedia promote the separation of the information of an application into two or more axes. Nonetheless, in practice, many applications do not separate these concerns and many of the current problems in the design, implementation, and maintenance of hypermedia applications arise because the applications are not divided according to the four-axes model. For instance, it is typical that the implementation of web sites is done directly in

HTML. HTML files contain, at the same time, content (the content of the elements of the HTML files), structure (HTML elements such as *TITLE, P, H1*), the navigation (the anchor and links elements *A*), and its presentation (such as FONT elements). It is difficult, therefore, to isolate each of the axes in order to reuse part of application for a different objective, or to maintain the application. For instance, to change the navigation of the application and break it into different nodes requires, first, to reorganize the content into different files, and second, to replace links in every affected file with the addresses of the new files. XML attempts to solve some of these problems by separating content and structure (in XML) from presentation (in XSL) and navigation (in XSL, and XLink).

1.1.2 Hypermedia Design Patterns

Many hypermedia design problems are ubiquitous and designers are likely to face them eventually. Common sense dictates that designers who face a new problem should not invent solutions from scratch; rather, they should take advantage of the knowledge acquired previously by others. A design pattern "describes a problem which occurs over and over again and then describes a solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice" [AIS77]. A design pattern attempts to collect experience from the expert to pass on to others in the field, avoiding reinvention.

Patterns are not unique to the hypermedia world. They were first used in architecture [Ale79] and recently used in software engineering design [GHJV, CS95, VCK96]. The first hypermedia design patterns were presented by Rossi [RSG97]; since then many more have been published. They range from generic "golden rules" [NN98] to specialized patterns for collaborative design [SS99].

Patterns are valuable hypermedia design artifacts for the following reasons [RSG97, Sch95, GPBV99]:

- Patterns improve communication within and amongst designers and developers.
- Patterns explicitly capture knowledge that experienced designers already understand implicitly.

- Patterns facilitate training of new designers.
- Patterns increase the quality of design.
- Patterns reduce the cost of design and implementation.

1.2 Problem Statement

The rapid growth of the World-Wide Web, and the corresponding growth in complexity of hypermedia applications has exposed the following needs:

- Applications are becoming bigger and more sophisticated. As a consequence it is more likely that they are not developed by a single individual. Communication between the different parts of the team becomes crucial.
- In many cases, the design and the implementation might be done by different teams. Design documents are the only means of communication between designer and implementor.
- Once the design documents have been created, it is desirable to verify that this design satisfies the requirements of the user before the application is implemented.
- The final user might contract a developer to create the application. A clear specification of the application can be the basis for a contract between both parties [Hol91].
- Maintenance is expected to be part of the lifetime of an application. The existence of design documents reduces the cost of maintenance.

In typical hypermedia developments these needs have not been completely satisfied [MD99, GGRS00]. The communication between developers is usually verbal and depends on the communication skills of the involved parties. The applications are described with a combination of very informal diagrams and textual descriptions that usually leave many details of the application unspecified; this type of description frequently leads to confusion and misunderstanding amongst the parties involved.

The graphical notations provided by methodologies such as OOHDM or RMM can be helpful to improve the communication between parties in a design. These problems are lessened, but not totally solved, mainly because these notations do not have very precise and well-defined syntax and semantics. As a consequence, different people might interpret the specifications in different ways.

For example, figure 1.1 depicts an RMM diagram taken from [ISB95]. This diagram shows an index of professors that satisfy the predicate *rank="associate"*. An index is a type of tour in which a node contains a listing of all the components of the tour with hyperlinks to their respective nodes.

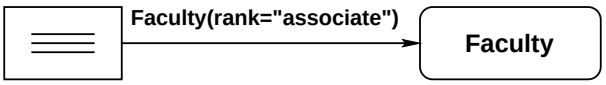


Figure 1.1: A guided tour of "associate professors" as depicted using the RMM notation.

The diagram clearly states that the components of the index tour are those in the set of *Faculty* which satisfy the predicate *rank="associate"*. RMM, does not describe, however, in detail the characteristics of each of its entities. The diagrams in the example in [ISB95] do not state that each element in *Faculty* has an attribute called *rank* and that one of the potential values of this attributes is *associate*. This information is implicit in the diagram. This diagram conveys information useful for a designer, and at the same time, it leaves many characteristics of the tour unspecified. For example:

- *Faculty* can be interpreted as a major academic unit of a university, or as a faculty member. The complete specification does not clearly define what *faculty* is. A designer will assume from hints in the diagrams—such as the predicate *rank="associate"*, in this case—that *faculty* refers to a professor.
- What attributes of each professor are shown in the index node; and in particular, which attribute is going to be used as an anchor for the link between the index and each of the professor's nodes? Is it the first name and last name of the professors, their email address, etc?
- In what order are the professors listed? Ascending? Descending? Which

attribute of each professor is used to sort them?

RMM does not have artifacts that can be used to answer the previous questions. OOHDM, on the other hand, even though more descriptive and rich in its expressiveness, also relies on semi-formal diagrams for the specification of the design. In both RMM and OOHDM, the specifications could lead to different interpretations by different people.

Another drawback of these semi-formal notations is the lack of a verification framework. There are no provisions to verify any property of the specification.

Formal models and methods that have been created with the purpose of the specification and verification of properties of hypermedia applications—such as [CDOS96, SFC98, WR98]—are too restrictive to be used to describe large hypermedia applications. Furthermore, these methods are not integrated into design methods, making it difficult to apply them.

The design of large hypermedia applications requires a notation for its specification that is unambiguous, precise and that does not constrain the implementation. This notation should be incorporated into the design process, in a manner than can be used in accordance with the current design methodologies. Finally, it should provide a formal framework in which it is possible to answer questions about the specification.

1.3 Proposed Solution

As a solution to the afore mentioned problems, we propose a formal specification language for hypermedia design called Hadez. Hadez is object oriented and is based on a data model that divides the specification of an application in three main parts:

- Conceptual schema. The conceptual schema specifies the characteristics of the underlying data used in the hypermedia application. It consists of two parts: a declaration of types and a declaration of relations amongst these types. The types are specified as given types, enumerated types, and type constructors (cross products, functions and classes). The relations create references between the different types and are later exploited in the creation of the structural and the perspective schema.

- Structural schema. The structural schema describes how the content of the application is combined into composites. These composites are not true content. They are, in database terminology, views on the conceptual schema. The Structural Schema is described with a sequence of conceptual schemas. The union of the conceptual schema and the structural schema results in the hyperbase schema.

- Perspective schema. A perspective allows a reader to perceive a composite. Different perspectives can exist for the same composite. As a result, the same composite can be perceived in different ways. A perspective behaves as an observer of a composite. A perspective has a state, and its state describes how the observed composite should be shown to the reader. Furthermore, a reader, by interacting with the perspective, can alter its state and, therefore, alter the way the perspective shows the composite.

The separation of a hypermedia into these three schemas is consistent with the four axes model and allows the designer to concentrate on different concerns at different times. The conceptual schema corresponds to the content axis, the structural schema to the structure axis, and the perspective schema plays the role of both, the navigation and presentation axes.

Hadez is based on the formal specification languages Z [Spi92a, BN92] and Z++ [Lan92], and is further extended with constructs oriented towards the specification of hypermedia.

Hadez has a formal syntax. This guarantees that a specification can be parsed in order to verify whether it is written according to the syntax rules of Hadez. A specification that passes this test is said to be syntactically correct.

A syntactically correct specification should then be verified to be complete. A specification is complete if it is syntactically correct and all the identifiers used in the specification are defined in the specification. Finally, if a specification is complete, it can by verified to be type-consistent. A specification is type consistent if none of its constructs violates the Hadez typing rules.

Reflecting its data model, a Hadez specification is composed of three parts, corresponding to each of the three schemas: conceptual, structural and perspective. Hadez provides a framework in which some questions about the application can be clearly stated and verified. And finally, Hadez incorporates the notion of hypermedia design patterns.

1.4 Contributions

This dissertation proposes Hadez, a formal specification language for the design of hypermedia applications, a data model in which Hadez specifications can be created, and a framework in which these specifications can be verified.

The primary contributions of this thesis can be summarized as follows:

- The formalization of a data model for hypermedia design.
- The definition of a hypermedia interface artifact, called *Perspective*, which serves as an interface between a hypermedia node and the user. The state of the perspective determines what attributes of the node the reader can see, and the potential actions that the reader can take.
- The syntax and semantics of Hadez, a formal specification language for hypermedia.
- A framework that supports the verification of a Hadez specification in order to answer questions about the specification.
- A framework, based on Hadez, for the formalization of hypermedia design patterns.

1.5 Related Work

Tompa proposed a model that separated content from structure, called the hypergraph data model. He established a clear separation between content and structure and allowed links to originate and end in a set of nodes. Finally, he proposed how the user could personalize her particular perception of a hypertext by creating views [Tom89].

The Trellis system [SF89] is a model for hypertext that is based on Petri nets. Nodes are composed of windows, content and buttons. The model provides the ability to create different views over the same content nodes. In [SFC98], Stotts and Furuta described a framework on which they could verify properties of hypertext applications based on the Trellis model. They defined a temporal logic

for hypertext called HTL*. HTL* is based on CTL* [CES86], and it adds quantifiers over potential browsing paths. Their work explains how questions about a given application can be translated into HTL* predicates. These predicates can then be verified with a model checker. Their system is used to specify and verify small, graph-oriented systems and does not seem to scale to large hypermedia systems, such as web sites.

In [SLHS93], Schnase et al. presented a model based on a semantic view of an object-oriented database. Casanova et al. introduced the notion of context nodes in their nested context model [CTL$^+$91, SRC95]. In this model, the browsing path that a reader follows affects the perception that reader has of a given hypertext document. Courtiat et. al explored the use of LOTOS for the specification and verification of time-dependent multimedia systems based on the nested context model [CdOdCC94, CO96, CDOS96, SSGC98]. They based their method on the nested-context model and applied reachability analysis to their specifications. Their main goal was to find inconsistencies in temporal constraints within the specification. These systems were demonstrated with very small applications and they do not seem to be easily applied to large hypermedia applications.

Garg proposed in [Gar88] a model for hypertext-based on first-order logic, but he did not include any semantics for navigation. His examples were simple graph and node type hypertext systems. In [dP95, dH97], Inverno et al. used Z to formalize the characteristics of a generic hypermedia model. Their main goal was to provide an unambiguous description of the semantics of a hypermedia system that could be used as a reference to compare actual hypermedia platforms. The Dortmund family of hypermedia models attempts to describe a variety of formal models for hypermedia [TD96]. Like many of its contemporaries, it incorporates the notion of content, node, and view. It describes these formal models using VDM. Labyrinth [DAP97] is a formal model for hypermedia; its main goal is to describe formally the characteristics of an application. The main contribution of this model is the inclusion of user permissions on the hyperbase and the ability to restrict content to groups of users. Each of these models contributed to the data model of Hadez.

Wang and Rada [WR98] used a graph-based model to specify the characteristics of hypermedia applications. Based on this model they created a framework in which to find structural inconsistencies in the specification. Their system,

again, was based on the notion of graph and node and did not support well large hypermedia applications built around databases.

Different methodologies appeared for the design of hypermedia systems, modeled after their software engineering counterparts. The Hypertext Design Model (HDM) [GPS91, GPS93] is a methodology oriented towards the development of large hypermedia applications. It emphasized the notion of perspective and it identified different types of links: structural, application and perspective links. A hypertext design consists of a schema definition and a set of instance definitions. The schema definition specifies a set of entity and link types; while the instance definitions specify the actual instances in the hyperbase. Several Hadez concepts, such as the notion of perspective, and type of instances and links and can be traced to HDM.

The Relationship Management Methodology (RMM) [ISB95] is similar to the entity-relationship model found in software engineering, but adapted to hypermedia. Entities and relations are present; entities can be divided in slices (where a slice is a subset of information about a given entity), which then can be presented to the user. Relations can be traversed by unconditional links or conditional links, where the link includes a predicate used to "find" the information sought. RMM defines a data model, called the Relationship Management Data Model (RMDM). This model uses a graphical notation, similar to the Entity-Relationship diagrams of software engineering, to describe a given hypertext design.

A successor of HDM, the Object-Oriented Hypermedia Design Methodology (OOHDM) uses an object-oriented approach to hypermedia design [SR94, SRB95b, RSLC95, SRB95a, Ros96]. OOHDM builds hypertext applications as navigational views of a hyperbase. Conceptual parts of a hypertext system are objects in an object-oriented database which are manipulated and composed to create different views, which become, eventually, hypertext nodes. The methodology uses Abstract Data Views (ADVs) [CL95] to attempt to formalize the design of the application. ADVs divide a hypermedia application into objects and their views. Objects are not viewable by the reader. Views provide, to the reader, a visual representation of their corresponding object (or objects). An ADV, within the hypermedia domain, is used to describe how each object is to be presented to the reader and how an application reacts to the events triggered by the reader; it describes the static and dynamic properties of an applica-

tion. OOHDM divides the design of an application in conceptual, navigational and abstract interface designs. Hadez separation of concerns is modeled after OOHDM. ADVs are direct ancestors of Hadez perspectives.

Araneus [AMM97, MAM+98a, MAM+98b] is a methodology for the design and implementation of data-intensive web sites. It is based on an entity-relationship model and divides the design of an application into its Hypertext Conceptual Design, which describes the data of the application) and its Hypertext Logical Design (which describes how the conceptual design is mapped to pages). The main goal of the Araneus Project is to provide an environment in which Web sites can be created following the Araneus design methodology and then implemented using a language called Penelope (based on ODMG[Cat96]). Contrary to Hadez, Araneus focuses on the implementation of the application. Hadez specifications, however, can be implemented using Araneus, making both systems complementary to each other.

The first hypermedia design patterns were presented by Rossi [RSG97]; since then many more have been published. They range from generic "golden rules" [NN98] to specialized patterns for collaborative design [SS99]. We present a review of hypermedia design patterns in [GC00]. There are several current attempts to integrate design patterns into design methodologies. Rossi et al. have been working on creating a system of patterns that can be incorporated into OOHDM [GRS97, LRS98a, Sch99, LRS98b]. At the same time, Garzotto and Paolini have been enhancing HDM with design patterns [PG99, GPBV99, GD99]. Discenza presented a graphical notation to describe the use of design patterns in hypermedia design [Dis99]; her approach is based on HDM and it is used to describe the Web Museum. of the National Gallery of Art in Washington, D.C., U.S.A.

1.6 Thesis Overview

In this thesis we present a formal specification language for hypermedia called Hadez. Chapter 2 presents a data model upon which Hadez is built. This model provides a strong separation amongst content, structure and presentation of a hypermedia application. This chapter introduces the concepts and nomenclature used in the rest of this document.

Chapter 3 describes the syntax and semantics of Hadez. In accordance with its data model, the specification is divided in three main parts: a conceptual schema—which describes the characteristics of the underlying data of the application; a structural schema—which describes how this data is structured; and the perspective schema—which describes how the user should perceive and interact with the application.

Chapter 4 is an example of a specification using Hadez. It describes the current web site of the National Gallery of Art, which is a complex hypermedia application and provides a good example of the features and expressiveness of Hadez.

Chapter 5 describes a framework for the verification of properties of a Hadez specification. Using the specification found in chapter 4, it argues how questions about the specification can be formalized and then verified.

Chapter 6 explains how Hadez can be further extended with the notion of design patterns.

Finally, in chapter 7 we summarize our work and discuss future research directions.

I am never content until I have constructed a model of the subject I am studying. If I succeed in making one, I understand; otherwise I do not.

Lord Kelvin
Baltimore Lectures on Molecular Dynamics
and the Wave Theory of Light, 1904

CHAPTER 2

Hadez Hypermedia Model

A data model is an abstract formalism that attempts to describe, mathematically, the properties of the information composing a given system. There is an implicit agreement that hypermedia applications can be modeled as graphs, in which the nodes are connected through links. This is the view that the reader has. A reader can view any hyperpage (node) and jump to other hyperpages by following hyperlinks. This model, albeit accurate from the point of view of the reader, is not the most appropriate from the point of view of the designer. A node is usually composed of information from different sources, all collated into a node. In effect, a node is a view that is created on top of the basic information that composes the systems.

The Hadez assumes that there is an underlying data repository, and that nodes are views created on top of it. These views can be presented in multiple ways to the reader and the reader can interact with the application.

In this manuscript, we will use the following notation. We will initially define it informally and later come back and define it precisely.

A *hyperdocument* is made of a collection of hyperpages. A *hyperpage* is the minimum unit of information that can be displayed to a reader at a given time. In an analogy to the World-Wide Web (WWW), a hyperdocument is a collection of HTML files; and hyperpages are equivalent to HTML pages. Each hyperpage is a *perspective* (or view) of a *composite*. A hyperpage, hence, is a visual representation of a composite. A composite is a collection of *content objects* and other composites. These content objects are the basic building blocks used to create the hypertext. We will refer to the collection of these objects and their composites as the hyperbase of the application.

Objects are at the core of the model. Composites are aggregates that combine them into higher level entities. These composites are then mapped into viewable entities (hyperpages) according to the characteristics of the run-time system in which they are browsed.

A hyperdocument, therefore, becomes one of multiple potential views of a hyperbase. The hyperbase contains a collection of content objects, relations and composite constructors. Composite constructors are specifications of how a composite should be created out of content objects and other composites. A composite can then be observed by a perspective in order to be presented to the reader as a hyperpage. The collection of these hyperpages creates a hyperdocument.

Hadez introduces the notion of type. In Hadez every content object has a type. Composite constructors specify the types of their component objects. Each composite itself has a type. Relations should specify the type of the objects they correlate. Links between hyperpages are typed. Perspectives have types. This guarantees that a specification of the application can be verified for type inconsistencies.

Because Hadez is a specification language, it is not concerned with how the hypermedia application is going to be implemented. It expects to be able to describe the characteristics of a hyperbase and its corresponding hyperdocuments, without constraining how it should be implemented.

2.1 A data model for hypermedia design

A hypermedia application is described with three different schemas: a conceptual schema, a structural schema and a perspective schema.

2.1.1 Conceptual Schema

The conceptual schema specifies the characteristics of the underlying data used in the hypermedia application. It consists of two parts: a declaration of types and a declaration of relations amongst these types. The types are specified as given types, enumerated types, and type constructors (cross products, functions and classes). The relations create references between instances of the different types and are later exploited in the creation of the structural and the perspective schema.

2.1.2 Structural Schema

The structural schema describes how the content of the application is combined into composites. These composites are not true content. They are, in database terminology, views on the conceptual schema. The Structural Schema is described with a sequence of conceptual schemas. The union of the conceptual schema and the structural schema result in the hyperbase schema.

2.1.3 Perspective Schema

A perspective allows a reader to perceive a composite. There can be different perspectives for the same composite. As a result the same composite can be perceived in different ways. A perspective behaves as an observer of a composite. A perspective has a state, and its state describes how the observed composite should be shown to the reader. Furthermore, a reader, by interacting with the perspective, can alter its state, and therefore, alter the way the perspective shows the composite. A perspective is specified with an Abstract Design Perspective (ADP) schema. A set of ADP schemas for a given hypermedia application A are known as the perspective schema of A.

2.2 Conceptual Schema

The conceptual schema of a hypermedia application is defined as:

2.1 Definition (Conceptual Schema) *The Conceptual Schema of a hypermedia application A is defined as a tuple:* $Schema_c(A) = \langle GT, TC, OI, R\rangle$ *where GT is a set of given types, TC is a set of type constructors, OI is a set of object instances, s.t.* $\forall o \in OI\, \exists t \in GT \cup TC \bullet o : t$*, and R is a set of relations s.t.* $\forall r : t_1 \times ... \times t_n \in R, t_1...t_n \in GT \cup TC \wedge \langle a_1, ..., a_n\rangle \in r \Leftrightarrow a_1...a_n \in OI$.

The conceptual schema of an application is the description of its types (either given—given types—or clearly specified—type constructors), a set of instance objects of those types and a set of relations between instance objects. In order to simplify our notation, we use $GT(A)$, $TC(A)$, $OI(A)$ and $R(A)$ to refer to the sets of given types, type constructors, object instances, and relations, respectively, of the conceptual schema of A.

2.2.1 Schema's Type Signature

Hadez is based on the concept of type consistency. Therefore every element of the application must have a precise type and the constructions of new elements must conform to type rules which cannot be violated. The type specification of an application is known as its schema's type signature:

2.2 Definition (Schema Type Signature) *For a conceptual schema* $Schema_c(A) = \langle GT, TC, OI, R\rangle$ *for a hypermedia application A, the schema's type signature* $TypeSig(A)$ *is defined as* $TypeSig(A) = PT \cup GT(A) \cup TC(A)$ *where PT is a set of predefined types.*

The type model of Hadez is based on Z++. A type defines a set (potentially empty, finite, or countably infinite) of values which a variable of such a type takes. We refer to the set defined by the type t as its domain, and we denote it as $domain(t)$. Hadez has various predefined types: the set of natural numbers $\mathbb{N}$, the set of positive integers $\mathbb{Z}$; and the set of reals $\mathbb{R}$.

The set $TC(A)$ is described using three different variants of type constructors: functions, cross products and classes.

2.2.2 Given types

As in Z (and Z++), types can be given. In this case, the inherent details of a type are not known, except that it is countable infinite. We can define the set of given types T as:

2.3 Definition (Set of Given Types)

$$T = \{t_i \mid i : 1..n, n \geq 0 \textit{ and } t_i \textit{ is a given type}\}$$

These types are provided by the specifier and are considered atomic, hence indivisible.

2.2.3 Enumerated Types

Enumerated types define collections of values that a variable can take. Each one of these values is considered atomic. Enumerated types define non-empty, finite sets, in which, as its name implies, each of the elements of the type is enumerated. Each one of its elements is identified by a unique name.

2.4 Definition (Enumerated Type) *An enumerated type t is:*

$$t = \{id_i \mid i : 1..n,\ n > 0\}$$

where id_i is a distinct identifier of the i-th element in the enumerated type.

2.2.4 Type constructors

More complex types can be defined through type constructors. A type constructor is an artifact used to create new types from combinations of other types. There are three type constructors in Hadez: *functions, cross products* and *classes*.

2.5 Definition (Properly defined Type) *A type constructor is properly defined if all the types it uses are properly defined. All atomic types are properly defined.*

In Hadez each type must be properly defined. As a consequence, whenever a type is defined, all the types used in its definition must be already defined. This restriction creates an acyclic directed graph in which the nodes are the types and the arcs show the dependencies in the definitions of the types.

Cross Products

2.6 Definition (Cross Product Constructor) *A type $t = t_1 \times t_2$ describes the set of values such that:*

$$domain(t) = domain(t_1) \times domain(t_2)$$

This can be further extended into a cross product of a finite number of types. Hence it is possible to define the cross products of three or more sets.

Functions

2.7 Definition (Function constructor) *A type $t_1 \rightarrow t_2$ defines a mapping between objects from set $domain(t_1)$ to the set $domain(t_2)$ such that for each element $v_1 \in domain(t_1)$ there is at most one element $v_2 \in domain(t_2)$ such that $v_1 \rightarrow v_2 \in domain(t_1 \rightarrow t_2)$.*

The types t_1 and t_2 can be of any type, including a cross product; as a consequence multiple parameters to a function are accepted.

Objects

Hadez is object-oriented. Objects offer important design features such as encapsulation, modularity, and extensibility. The underlying data in an application is a collection of classes instances. A class is defined as:

2.8 Definition (Class) *A class C is defined, recursively, as $C = \{P, l_i : t_i, i \in 1..n\}$, where l_i is a field of type t_i within C, and P is also a class.*

Hadez defines a superclass C_s with no attributes from which all classes inherit. Only single inheritance is allowed. Note that there is no distinction between methods and attributes in the data model of Hadez. The reason is that Hadez applications do not modify the underlying data; as a consequence, both attributes can be treated as functions with no parameters that return a given value.

2.2.5 Relations

Relations are the way in which different objects in the conceptual schema are related amongst themselves. Hadez benefits from the ample range of variants

of relations in Z's mathematical toolkit. A relation between two sets, A and B, is a subset of the cross product of those sets. Furthermore, a function can be considered as a special case of a relation.

Table 2.1 lists the types of relations and functions used in Hadez.

$\leftrightarrow$	Binary relation
$\rightarrow$	Total function
$\nrightarrow$	Partial functions
$\rightarrowtail$	Total injection
$\nrightarrowtail$	Partial injection
$\twoheadrightarrow$	Total surjection
$\nrightarrow\!\!\!\!\twoheadrightarrow$	Partial surjection
$\rightarrowtail\!\!\!\!\twoheadrightarrow$	Bijection

Table 2.1: Different types of relations used in Hadez.

All of them are formally defined in chapter 4.2 of [Spi92a]. Relations form the basis for more complex data types. For example, a sequence of elements of type t is a partial function from integers to t; a set of elements of type t is a total function from t to $\{true, false\}$.

2.3 Hyperbase

2.9 Definition (Hyperbase) *A hyperbase for an application A is defined as a tuple:*

$$Hyperbase(A) = \langle Schema_c(A), OI, CS \rangle$$

where OI is a set of object instances, CS is a set of composite schemas.

Formally, the set of object instances is defined as:

2.10 Definition (Set of Instances) *The set of instances IS of an application A is:*

$$IS(A) = \{x \mid \exists t \in TC(A) \bullet x : t\}$$

A composite is, informally, a set of object instances and other composites. It allows the definition of higher-level entities from the object instances in OI. A

composite schema is a description of how to instantiate a composite from a set of parameters.

From the point of view of the reader, an application is a collection of composites. Each of these composites is a collection of other composites or object instances. This aggregation creates higher-level objects from the low-level object instances in the hyperbase.

In order to illustrate these definitions, assume a simple hypermedia museum that shows a collection of paintings. The hyperbase contains one type: painting, and the set of instances is, say, 10 different painting objects. The application might specify that only the oldest 5 paintings are to be presented, in the order of their creation date. In this case, a composite is defined to be a sequence of paintings that satisfies the following conditions: that it lists the 5 oldest paintings, and that they are ordered by their creation date. Note that no new content has been created. The composites in effect serve as "content wrappers" that simplify the specification of the application.

Composites do not actually reside in the hyperbase. Since they provide no new content to the application, they are dynamically instantiated. The hyperbase, instead, contains descriptors for each one of the composites that indicate how each composite is to be instantiated. These descriptors are called composite schemas in Hadez. We will refer to the set of composite schemas of an application A as CS(A).

A composite schema, besides describing how to instantiate a composite, describes its type signature and the type signature of its components.

2.11 Definition (Composite) *A Composite C in the application A is defined recursively as a set* $C = \{c_1 : t_1, ..., c_n : t_n\}$ *(potentially empty) where* c_i *is a composite instance of an object instance of type* t_i*; such that* $t_i \in OI(A) \cup CS(A)$

In other words, a composite in the application A is a set of composites or object instances whose types are defined in the hyperbase of the application A.

2.12 Definition (Set of Composites) *For an application A, its set of composites* $\mathbb{C}(A)$ *is the set of all the composites instantiable from the hyperbase of A (Hyperbase*(A)*).*

That is, $\mathbb{C}(A)$ is the set of any possible composite that can be instantiated by an application A.

We can define the following relation amongst composites:

2.13 Definition (Containment $\subseteq$, $\subset$) *Given two composites:* $C = \{C_1, ..., C_n\}$, *and* C_j*:*

$$C_j \subseteq C \Leftrightarrow \begin{cases} C = C_j & \text{or} \\ \exists k \in 1..n \quad s.t. \quad C_k \subseteq C_i \end{cases}$$

$$C_j \subset C \Leftrightarrow \quad C_j \neq C \quad \wedge \quad C_j \subseteq C$$

C_j is contained in C if either $C_j = C$, or C_j is contained in one of the children of C. $C_j \subseteq C_i$ indicates that C_j is a member of C_i or of one of its children. In order to avoid infinite recursion we need to force the following constraint:

2.14 Observation (No self-containment) *A composite cannot contain itself:*

$$\forall C_i \in \mathbb{C}, \quad C_i \not\subset C_i$$

Corollary 2.15 ($\subseteq$ is a partial order relation on C)

The containment relation ($\subset$) can be represented with a hierarchical acyclic graph in which the internal nodes are the composites, the leaves the instances and the arcs mono-directional representations of the relation (see figure 2.1).We will refer to this graph as the structural graph of a hyperbase and, by extension, of an application. This graph is a visual representation of the partial order described in corollary 2.15. Because one composite can be included in more than zero or more composites, the graph is not a tree.

We now proceed to illustrate these definitions with the following example. Assume we need to create a hyperbase for a tiny digital museum. For the sake of simplicity assume that we have two distinct types: *painting* and *artist*. Each painting has attributes such as image, date of creation, description, technique, etc. For each artist we have a short biography, birth date, nationality, etc. The artists and the painters are related by the relation *PaintedBy* : *painting* $\leftrightarrow$ *artist*.

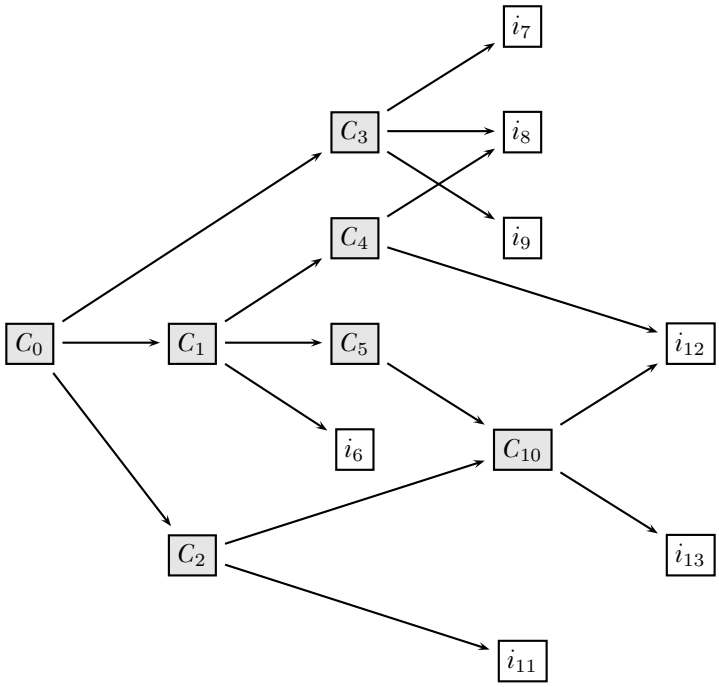


Figure 2.1: An example of a structural graph of a hyperbase. The shaded nodes denote composites

The hyperbase is going to be composed of the following object instances: the paintings *Naval Battle* by *Kandinsky*; *Café Terrace*, and *Sunflowers* by *Van Gogh*; and *Metamorphosis III* by *Escher*.

We want to provide three different views: a painting view, which displays the painting's attributes and some of its author's attributes; an artist view, which displays the author's attributes plus a listing of all the painting names; and a main view, which provides direct access to all the paintings, and to all the artists.

$$
\begin{aligned}
GT(Museum) =& \{artist, painting\} \\
OI(Museum) =& \{\text{“Van Gogh”, “Escher”, “Kandinsky”} : artist, \\
& \quad \text{“Cafe Terrace”, “Metamorphosis III”,} \\
& \quad \text{“Naval Battle”, “Sun Flowers”} : painting\} \\
R(Museum) =& \{PaintedBy : painting \leftrightarrow artist\} \\
CS(Museum) =& \{ArtistComposite, PaintingComposite, MainComposite\}
\end{aligned}
$$

Figure 2.2: Simplified Hyperbase of the Museum.

$$
\begin{aligned}
PaintedBy =& \{\text{“Cafe Terrace”} \mapsto \text{”Van Gogh”}, \\
& \quad \text{“Sun Flowers”} \mapsto \text{“Van Gogh”}, \\
& \quad \text{“Naval Battle”} \mapsto \text{“Kandinsky”}, \\
& \quad \text{“Metamorphosis III”} \mapsto \text{“Escher”}\}
\end{aligned}
$$

Figure 2.3: The details of the relation *PaintedBy*.

Figure 2.2 shows an abbreviated description of the hyperbase for the digital museum. Figure 2.3 details the elements of the relation *PaintedBy*.

The type signatures of the composite schemas are defined in figure 2.4. An *ArtistComposite* is composed of an *artist* and a set of *paintings* (as they are related by *PaintedBy*. The *PaintingComposite* on the other hand, contains a *painting* and its creator, an object of type *artist*.

$$
\begin{aligned}
\mathit{ArtistComposite} &\triangleq \{A : \mathit{artist}, \mathit{Paintings} : \mathbb{P}\,\mathit{painting}\} \\
\mathit{PaintingComposite} &\triangleq \{A : \mathit{artist}, P : \mathit{painting}\} \\
\mathit{MainComposite} &\triangleq \{\mathit{ArtistsComp} : \mathbb{P}\,\mathit{ArtistComposite}, \\
&\qquad \mathit{PaintingsComp} : \mathbb{P}\,\mathit{PaintingComposite}\}
\end{aligned}
$$

Figure 2.4: The type signature of the composites in Museum.

MainComposite contains two different components: *ArtistsComp* which is a set of all the *ArtistComposite* in the hyperbase, and *PaintingsComp*, which is a set with all the *PaintingComposite* in the hyperbase. The resulting composites are shown in figure 2.5.

$$
\begin{aligned}
\mathit{PaintingComposite}_1 &= \{\text{“Cafe Terrace”, “Van Gogh”}\} \\
\mathit{PaintingComposite}_2 &= \{\text{“Sun Flowers”, “Van Gogh”}\}, \\
\mathit{PaintingComposite}_3 &= \{\text{“Naval Battle”, “Kandinsky}\}, \\
\mathit{PaintingComposite}_4 &= \{\text{“Metamorphosis III”, “Escher”}\}, \\
\mathit{ArtistComposite}_1 &= \{\text{“Van Gogh”, “Sun Flowers”, “Cafe Terrace”}\}, \\
\mathit{ArtistComposite}_2 &= \{\text{“Kandinsky”, “Naval Battle”}\}, \\
\mathit{ArtistComposite}_3 &= \{\text{“Escher”, “Metamorphosis III”}\}, \\
\mathit{MainView} &= \{\{\mathit{ArtistComposite}_1, ..., \mathit{ArtistComposite}_3\}, \\
&\qquad \{\mathit{PaintingComposite}_1, .., \mathit{PaintingComposite}_4\}\}
\end{aligned}
$$

Figure 2.5: The different composites created from the Conceptual Schema of Museum.

Figure 2.6 shows the composite graph corresponding to the Museum application.

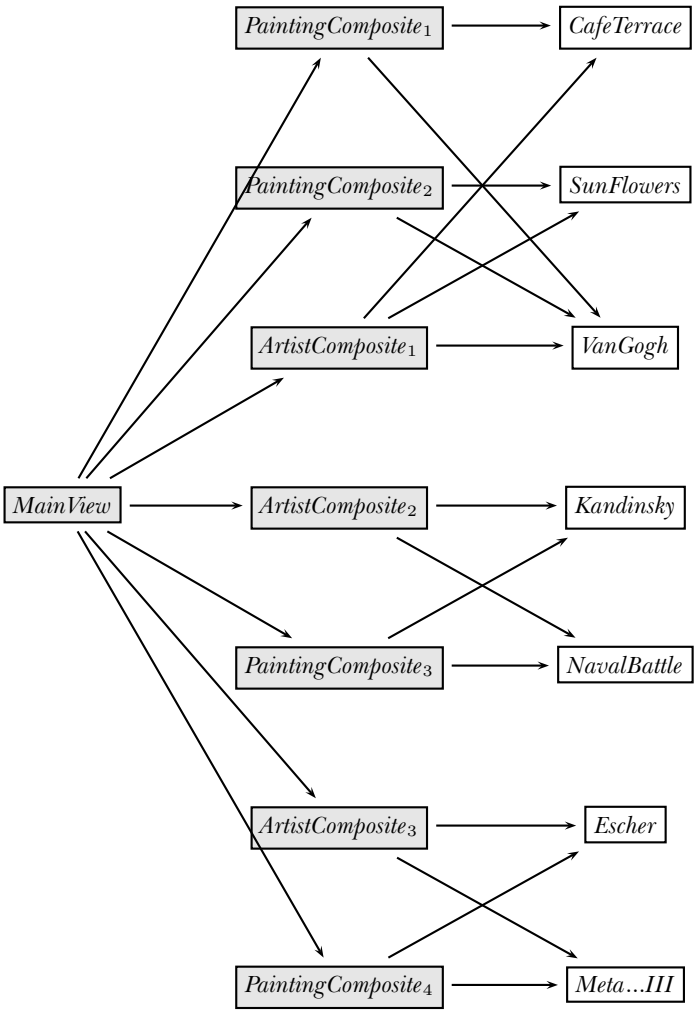


Figure 2.6: Visual representation of the composites.

2.4 Structural Schema

In a real application it is not feasible to enumerate each of the composites the way that was done in the previous section for the Museum application. As we have described before, a composite is instantiated using a composite schema.

2.16 Definition (Composite Schema) *A Composite Schema C_s of an application A is defined as $C_s = \langle\{p_1 : t_1, ..., p_n : t_n\}, \{c_n + 1 : t_{n+1}, ..., c_{n+m} : t_{n+m}\}, \langle P_1, ..., P_n\rangle\rangle$ where $p_i : t_i$ is an instantiation parameter of type t_i, c_i is free variable corresponding to a component of type t_i, $\langle P_1, ..., P_n\rangle$ is a sequence of predicates on the parameters p_i bounding each c_i to another composite $c \in \mathbb{C}(A)$ or to an object instance $o_i \in OI(A)$.*

A composite schema defines a composite type. The composite schemas indicate how to create a composite given a set of instantiation parameters. Each of the types $t_1, ..., t_{n+m}$ is either part of $TC(A)$ or a composite type.

The set of composites schemas is known as the structural schema of an application.

2.17 Definition (Structural Schema) *The structural schema of an application A, denoted by $Schema_s(A)$, is a set of composite schemas created for the application A.*

When we described the conceptual schema of the Museum application, we listed the type signatures of the composites, and enumerated all the different composites that could be created from the Museum's hyperbase. Figure 2.7 shows the structural schema for the Museum application. *PaintingCompositeSchema* has one parameter: a painting p, and one component: the artist a who created it. Note how the relation *PaintedBy* is used to bind the value of the p based on the value of a. *ArtistCompositeSchema* is similar to *PaintingCompositeSchema* but it takes as a parameter an artist a and it has as a component a set of *paintings*. The schema *MainComposite* is a bit different; it does not require any parameters and it has composites as its components. The predicate $A_c = ArtistCompositeSchema(a)$ is always true and it binds the value of A_c to a composite which is instantiated with the composite schema *ArtistCompositeSchema* that takes a as its instantiation parameter. The use of the composite schema's name within a predicate is known as a composite constructor. In *MainCompositeSchema* one *ArtistCompositeSchema* is

$$
\begin{aligned}
\mathit{PaintingCompositeSchema} =& \langle \{p : \mathit{painting}\}, \{a : \mathit{artist}\}, \\
& \langle a \mapsto p \in \mathit{PaintedBy} \rangle \rangle \\
\mathit{ArtistCompositeSchema} =& \langle \{a : \mathit{artist}\}, \{P : \mathbb{P}\, \mathit{painting}\}, \\
& \langle \forall p : P \bullet a \mapsto p \in \mathit{PaintedBy} \rangle \rangle \\
\mathit{MainCompositeSchema} =& \langle \langle\rangle, \{\mathit{Paintings} : \mathbb{P}\, \mathit{type}(\mathit{PaintingCompositeSchema}), \\
& \mathit{Artists} : \mathbb{P}\, \mathit{type}(\mathit{ArtistCompositeSchema})\}, \\
& \langle \forall a \in \mathit{domain}(\mathit{artist}) \exists A_c \in \mathit{Artists} \bullet \\
& \qquad A_c = \mathit{ArtistCompositeSchema}(a), \\
& \forall p \in \mathit{domain}(\mathit{painting}) \exists P_c \in \mathit{Paintings} \bullet \\
& \qquad P_c = \mathit{PaintingCompositeSchema}(p) \rangle \\
& \rangle
\end{aligned}
$$

Figure 2.7: Structural Schema of Museum.

instantiated for each element in the domain of the type *artist*; similarly, for each element in the domain of the type *painting*, a *PaintingCompositeSchema* is instantiated.

2.5 Perspective Schema

As we described in section 2.3, a hyperbase is not a navigable hypermedia. The hyperbase only describes the application's data and how it can be interrelated through composites into higher level entities. We define a hypermedia as a navigable view on the hyperbase. There are three aspects involved in creating a hypermedia from a given hyperbase:

1. Pagination. We must describe how the composites of the hyperbase are translated into hyperpages.

2. Selection. For a given hyperpage the hypermedia might present only a subset of the attributes of each of its composites.

3. Linking. A link has two components: a source and a destination. The source of a link has one attribute: its label (the text that appears in the an-

chor of the link). The destination link has two attributes: the destination hyperpage and a label within this hyperpage (potentially empty). Both source and destination values depend upon the value of the composite. Hadez indicates what links are created, what their source labels are, and what their destination hyperpages (and labels) are.

2.5.1 Pagination

The pagination of a composite is the specification of how the composite is to be converted into one or more hyperpages. For instance, a composite can be mapped into one or more hyperpages.

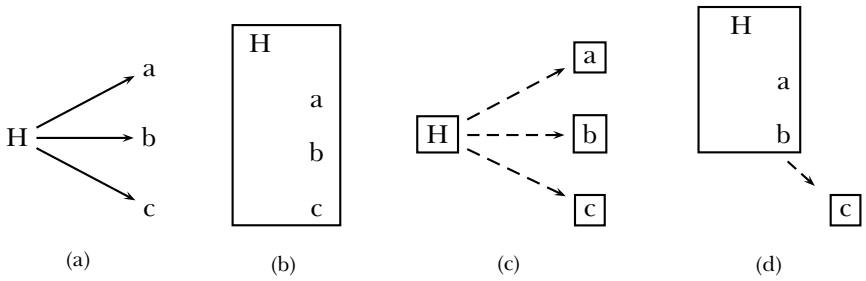


Figure 2.8: The same composite H is presented in three different ways. (a) shows its structural graph paginated in three different ways: into a single page, into two pages, into four pages. The dashed lines depict hyperlinks.

In order to illustrate the concept of pagination, consider a very simple composite H composed of a, b and c. The structural graph of this composite is depicted in figure 2.8 (a). The same composite can be paginated in different ways: in 2.8 (b) the composite and its components are presented in the same hypernode; in 2.8 (c) each component is separated in its own hyperpage and H serves an access point for each; finally, in 2.8 (d), H, a, and b are all presented in one hypernode, while c is presented in a different one. In these figures, the hash lines depict hyperlinks in the direction of its arrow. Note that the inclusion relationship between different composites results in structural hyperlinks.

The separation of pagination from the hyperbase allows a very flexible system in which the same composite can be presented in several ways, depending

on the characteristics of the final hypermedia :

1. The characteristics of the run time system. In some rudimentary hypermedia systems—such as Acrobat reader—the final application has a linear nature. A composite might be displayed in single, very long hyperpage. If the system is highly interactive, such as multimedia CD-ROMs, it might be desirable that no hyperpage is longer than certain size, hence each composite will be broken into multiple hyperpages.

2. The intended use. Some hypermedia applications present two views of the same information, one for regular browsing, and the other intended for printing. In the former the composite is broken into several pages, while in the latter it is mapped into only one. A typical example is help systems, that should be printed in paper and also presented on the WWW.

3. The location where the composite is included. Sometimes the same composite is shown in two different places, and presented in each case in different ways; the pagination might be different in each case, depending on which path the reader has followed to reach the current node.

4. Its intended audience. For people with very slow links it might be desirable to break the composite into small chunks of information, so they are downloaded as fast as possible.

By separating the creation of pages from the specification of the structure of the application, it is possible to create different navigable views of the same hyperbase.

Figure 2.9 shows the structural graph in figure 2.1 (page 26) that is paginated. In this example, composite C_{10} can be reached in two different ways: from C_2 and from C_5. The hypermedia paginates C_{10} in two different ways depending on which path is used to reach the page.

2.5.2 Selection

A composite (or an object) is usally composed of several components or attributes. A given hypermedia might not necessarily present all of them. Which attributes of the composite are shown to the reader might depend upon:

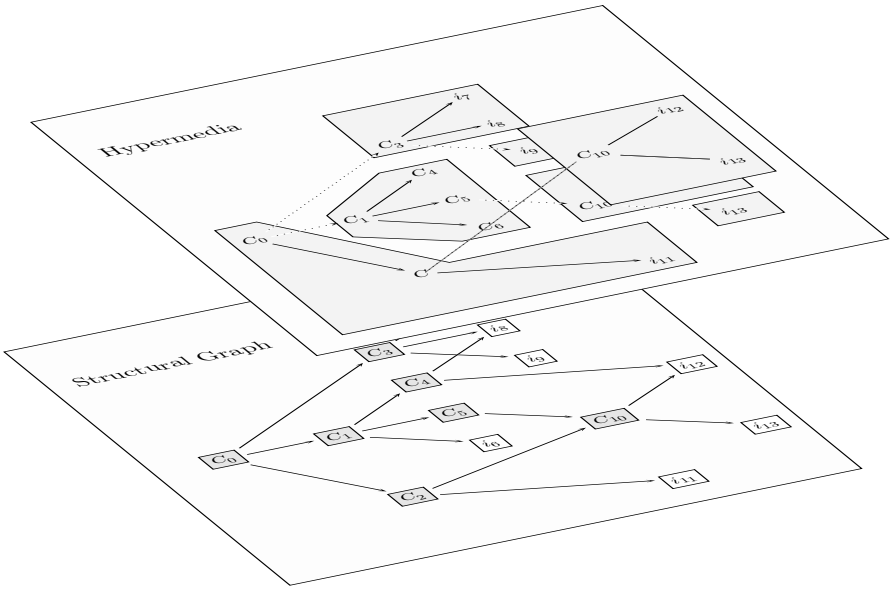


Figure 2.9: A pagination for the structural graph in figure 2.1. C_{10} is paginated in two different ways depending on the path followed to reach it.

1. Design decisions. The designer determines which are the attributes of the composite to be shown in a given hyperpage.

2. Reader interaction. The run-time system might be able to receive instructions from the reader indicating which attributes should be shown or hidden.

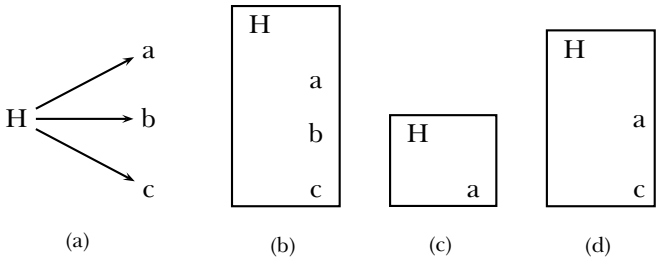


Figure 2.10: Different attributes of the composite *H* are selected in three different hyperpages.

Figure 2.10 graphically depicts the effect of selection on one composite. In 2.10 (b) all its components are selected, in 2.10 (c) only *a* is selected, while in 2.10 (d) only *b* is left out.

2.5.3 Linking

For every link in the hypermedia, it is necessary to specify its source label and its destination. Both are dependent on the content of the current composite. For example, assume that $PaintingComposite_i$ from the *Museum* application is presented in one page, which lists details of the painting (such as its name), and provides a link to a different hyperpage that shows the details of the author. A good choice for the links label is the name of the author of the painting. The destination of the anchor will be the page where the composite $ArtistComposite_j$ (which corresponds to the creator of the painting) is presented.

These three aspects—pagination, selection and linking—are modeled using regions and Abstract Design Perspectives.

2.5.4 Region

When a hypermedia application is displayed to the reader, it usually occupies a fixed area of the screen. We will refer to this area as the viewport of the run-time system. In general, a viewport displays information from the hyperbase and interacts with the reader. The reader is able to scroll through the information displayed, follow hyperlinks or change the state of the viewport. The viewport, however, can be considered a group of small regions which collaborate together. Each region, in itself, can be composed of smaller regions. A viewport, therefore, is a region that does not share the viewport with any other one. Regions are used as place-holders for the hyperbase content.

Figure 2.11: A viewport is broken into regions. Each region is responsible for displaying a composite.

2.5.5 Perspectives

A perspective is responsible for mapping a composite into a region. A perspective indicates which attributes of the composite are selected to be shown to the reader. It also describes the links to be displayed within the current region.

A perspective is an observer of a composite in the hyperbase (the perspective's observed composite), and is an interface between the reader and its observed composite. It has its own state and it cannot alter its observed composites. For each of its potential states it specifies which attributes from the composite it should display. It can be composed of several other composites

which remain hidden from the outside, preserving the principle of encapsulation. A perspective acts independently of other perspectives and communicates with them only through messages. Messages are divided in three types: external, internal and output. An input message can be generated by the reader or by another perspective in another region. Internal messages are generated by the perspective and they should be handled internally by the perspective itself or its component perspectives. Output messages are handled by other perspectives in other regions or by the run-time system. The main characteristics of perspectives can be summarized as follows:

- A perspective has a collection of local attributes that define a set of states. This set of states can be infinite.
- A perspective can change its state only through input messages. Input messages are generated by the reader, by the run-time system or by other regions within the current viewport. A change in a perspective from one state to another is known as a transition.
- A transition can trigger an output message which is intended for perspectives in other regions.
- The perspective specifies, for each state, what parts of the observed composite are shown to the reader. The set of attributes shown to a reader in a given state is known as the *current selection* of the perspective. A transition of the perspective, hence, might determine a change in the current selection. The perspective, therefore, has the ability to present different information to a reader at different times.
- A perspective can be composed by a finite number of perspectives. The communication between the different component perspectives is done through internal messages.

A perspective is defined as[1]:

2.18 Definition (Perspective) *A perspective P is a tuple $P = \langle S, s_0, s_h, C, M, \Psi, V \rangle$ where S is a set of states; s_0 is the starting state for the perspective s.t. $s_0 \in S$; C is*

[1]Section 5.4 (on page 120) is dedicated to the formalization of perspectives.

a composite being observed by the region; M is a set of messages that the region either handles or generates; Ψ is a relation of the form $\Psi \subseteq S \times M \times S$; and V is a relation of the form $V \subseteq S \times \mathcal{A}(C)$, in which $(s_i, c_j) \in V$ iff component c_j of C is visible at state s_i. There exists one state $s_h \in S$ s.t. such that C is not visible; we refer to this state as the hidden state of the perspective.

This section provides a brief overview of their properties. The perspective's state has two main purposes. On one hand it determines how it renders its observed composite C. The perspective's rendering of C can be modified by changing the state of the perspective. On the other hand, a perspective's state determines which messages $m \in M$ it can handle.

M is partitioned into three disjoint sets: input, output and internal. We refer to each one of these sets as $input(M), output(M)$ and $internal(M)$, respectively.

Examples of input messages that a perspective can receive are:

- Display anchor. A perspective can define a finite set of destination link labels (hyperlink destinations). These links define subareas within the current perspective. A *display anchor* message indicates to the perspective that the reader has followed a link whose destination is within the perspective.
- Reader generated messages. A perspective defines a list of messages that the reader can send to it. They alter the current state of the perspective and, optionally, generate an output message to another perspective in another perspective. An example of these messages is the selection of the source anchor of a link. Hadez uses the notion of buttons. A button is a label in the current perspective that, when selected by the reader, generates an specified message.
- Perspective-to-perspective messages. A perspective can receive messages generated by other perspectives within the current viewport.

On the other hand, the output messages are categorized as follows:

- Link selection. A link within the perspective has been selected by the reader and the perspective should be replaced by the destination perspective. These messages are handled by the run-time system.

- Perspective-to-perspective messages. These messages, as their input counterparts, are meant to be handled by another perspective in the current viewport.

There must always exist a transition $(s_i, m, s_j) \in \Psi$ for each $s_i, s_j \in S$ and $m \in M$. This guarantees that the perspective can always handle any message it receives.

To clarify these concepts, we will proceed to describe a perspective for the Museum application. The perspective will be responsible for observing a composite $PaintingComposite_n$ as defined in figure 2.3. The perspective will show the following attributes: the painting's name, a picture of it, and the name of its painter; it will receive two messages "Show description" and "Hide description" which will force the perspective to show or hide the description of the painting; the name of the painter links to another perspective which observes the corresponding *ArtistComposite*.

The painter's perspective messages that the perspective can generate or handle are listed in figure 2.12.

input message:	SELECT_LINK_AUTHOR, SHOW_DESCRIPTION, HIDE_DESCRIPTION
output messages:	GOTO_AUTHOR_COMPOSITE
internal actions:	none

Figure 2.12: Messages signature of perspective *Painting*.

The transitions of the perspective are described in figure 2.13.

```
PRECONDITIONS:
                    show(Painting.name)
                    show(Painting.picture)
                    show(Painting.a.Name)
SELECT_LINK_AUTHOR:
                    SEND MESSAGE
                        GOTO_AUTHOR_COMPOSITE ArtistComposite(Painting.a)
SHOW_DESCRIPTION:
                    show(description)
HIDE_DESCRIPTION:
                    hide(description)
```

Figure 2.13: Transitions of *Painting*.

The "visibility" of each attribute of the observed composite can be changed with the predefined predicates *show* and *hide*. The state of the perspective is determined by which attributes have their "visibility" on.

The *PRECONDITIONS* section specifies the start state of the perspective. The *SELECT_LINK_AUTHOR* indicates that, whenever that messages is received, an output message, *GOTO_AUTHOR_COMPOSITE* is sent to the run-time system. This message takes a parameter: the destination perspective and its corresponding observed composite *Painting.a*. This means that the current perspective is to be replaced by a perspective that observes the composite *ArtistComposite* instantiated with the attribute *a* (the author of the painting) of the current composite.

Building complex Perspectives

Perspectives can be combined to specify more complex regions. There are three ways in which perspectives can be combined:

- Inheritance. A perspective can extend the functionality of another perspective.
- Aggregation. A perspective has other perspectives as its components. The component perspectives describe subregions within the current region of the perspective. A region like the one depicted in figure 2.11 can be modeled by a group of aggregated perspectives.
- Composition. A perspective can be created by composing two or more perspectives. Neither perspective is inside the other.

Abstract Design Perspective Schema

In Hadez the perspective is specified with an Abstract Design Perspective schema (ADP schema, or ADP for short). Figure 2.14 shows the general form of an ADP. The details of ADPs and their relationship to perspectives will be described in chapter 3.

Perspective Schema

A perspective schema can be defined as:

ADP Name [: Ancestor] **Observes** Composite:Type
Declaration: $attrname_1 : type_1, ..., attrname_n : type_n$,...,
Block $block_1$**:** *block specification*
Block $block_2$**:** *block specification*
...
Preconditions: *predicates*
Invariants: *predicates*
Message $event_1$**:** $post - condition\ predicate$
Message $event_2$**:** $post - condition\ predicate$
...
End Name

Figure 2.14: An Abstract Design Perspective specification schema.

2.19 Definition (Perspective Schema) *A perspective schema of an hypermedia application A, denoted by* $Schema_p(A)$*, is defined as a tuple* $Schema_p(A) = \langle \mathbb{P}, \mathbb{P}_0 \rangle$ *where* $\mathbb{P}$ *is a set of perspectives on composites of* $Schema_c(A)$ *and* $\mathbb{P}_0$ *is a non-empty set of start perspectives.*

The perspective schema defines a set of the perspectives. Each of these perspectives defines a region. The set $\mathbb{P}_0$ defines a group of perspectives that are "entry points" to the hypermedia application; for each of these perspectives, its corresponding region is the entire viewport. An entry point page is intended to become the first page visited by a reader.

2.6 Summary

The data model of Hadez has three components:

- Conceptual schema. The conceptual schema specifies the characteristics of the underlying data used in the hypermedia application. It consists of two parts: a declaration of types and a declaration of relations amongst these types. The types are specified as given types, enumerated types, and type constructors (cross products, functions and classes). The relations create references between the different types and are later exploited in the creation of the structural and the perspective schema.

- Structural schema. The structural schema describes how the content of the application is combined into composites. These composites are not true content. They are, in database terminology, views on the conceptual schema. The Structural Schema is described with a sequence of conceptual schemas. The union of the conceptual schema and the structural schema results in the hyperbase schema.

- Perspective schema. A perspective allows a reader to perceive a composite. There can be different perspectives for the same composite. As a result, the same composite can be perceived in different ways. A perspective behaves as an observer of a composite. A perspective has a state, and its state describes how the observed composite should be shown to the reader. Furthermore, a reader, by interacting with the perspective, can alter its state and, therefore, alter the way the perspective shows the composite. A perspective is specified with an Abstract Design Perspective (ADP) schema. A set of ADP schemas for a given hypermedia application A are known as the perspective schema of A.

The separation of a hypermedia application into these three schemas allows the designer to concentrate on different concerns at different times. It also isolates the description of the application domain data from the way it is organized and presented to the reader. It is also possible to design applications that use the same underlying data, but results in an entirely different hypermedia application, in which the information is presented in a different way.

The strict separation of concerns of the Hadez data model supports reuse, portability, run-time system independence and low maintenance and serves as the basis for the Hadez specification language described in the next chapter.

[The grand book of the universe] is written in the language of mathematics, and its characters are triangles, circles and other geometric figures, without which it is humanly impossible to understand a single word of it; without these, one wanders in a dark labyrinth.

Galileo Galilei
The Assayer (Il saggiatore), 1623

CHAPTER 3

A Specification Language for Hypermedia

Formal specifications use a mathematical notation to describe, in a precise and unambiguous way, the properties which an information system must have without unduly constraining the way in which those properties are achieved [Spi92a]. Furthermore, questions regarding the characteristics of the final application can be answered with confidence without resorting to analyzing the final entangled application; and formal specifications are unambiguous, contrary to informal, textual specifications of an application.

Hadez is a specification language for hypertext design. Hadez is object-

oriented and it is based on the Hadez data model (described in chapter 2). Hadez is based on the formal specification languages Z [Spi92a, BN92] and Z++ [Lan92], and further extended with constructs oriented towards the specification of hypermedia.

3.1 An overview of Hadez

Following the model described in chapter 2, Hadez is divided into three parts: conceptual schema, structural schema, and perspective schema.

3.1.1 Conceptual Schema

The conceptual schema of Hadez is a collection of given types, type constructors, classes definitions, relations and instances. This part of Hadez is very similar to a Z structural specification (as opposed to a behavioral one). Hadez allows the declaration of classes that can be defined using single inheritance.

3.1.2 Structural Schema

The structural schema is a collection of Hadez composite schemas. A composite schema describes how to create a composite from other composites or from data from the conceptual schema. As described in section 2.4 (on page 30), a composite schema is composed of three main parts: its parameters section, which specifies what composites or data are required to instantiate the composite; a sequence of free variables; and a set of predicates that bind those free variables to instances in the hyperbase or to other composites.

A composite schema in Hadez looks like a Z schema, with two main differences: in Hadez the name of the composite is preceded by the Greek letter Ξ; and, composites can inherit their characteristics from other composites. This inheritance is simply a syntactic facility that allows reuse of the characteristics of other composites.

Another feature of Hadez is ability to define generic composite schemas. A generic composite schema's main difference from a typical composite schema is its use of generic types. A generic type is used like a given type (that is, its details are unknown), and it must be instantiated before it can be used (using a

generic type instantiation). The main objective of generic composite schemas is, like templates in C++, to avoid repetitive constructs and promote reuse of the specification.

3.1.3 Perspective Schema

The perspective schema is composed of a sequence of abstract perspective schemas (ADP). An ADP serves three main purposes: 1) it specifies how the composite should be broken into pages; 2) it indicates which attributes of the composite should be presented to the reader, and how they are organized within the hypernode; and 3) it specifies what attributes of the composite should be hyperlinked to other composites.

A perspective is not a static entity. The user can change its state. An ADP specifies how the perspective reacts to user requests. The state of an ADP determines what parts of the composite the ADP should show at that particular state.

ADPs can be composed into more sophisticated ones by using one of the following constructs:

- Aggregation. One or more composites are embedded into another composite.
- Inheritance. An ADP inherits its characteristics from another ADP.
- Parallel composition. One or more ADPs are composed into a more complex ADP.

In the following sections we describe in detail the characteristics of Hadez and each of these schemas.

3.2 Hadez language

A Hadez specification consists of a collection of statements that can be interleaved with textual descriptions. The statements introduce the different components of the application, while the informal text provides a comment on the meaning of the statements. Hadez extends the syntax and semantics of Z in

order to accommodate the requirements of the specification of hypermedia applications.

The majority of the constructs of Hadez use predicates to specify properties of the application. These predicates are written in Z—as defined in the Z proposed ISO standard [BN92]. Furthermore, Hadez takes advantage of the rich mathematical toolkit of Z. Hadez borrows from Z++ the notion of classes and subtyping, as Z does not implement object-oriented features.

The conceptual schema of an application is defined mainly in Z and the specification of objects is made in a simplified version of Z++. Hadez does not require the full set of features of Z++ because its objects are simpler: multiple inheritance is not permitted, the details of the methods of an object are not a concern of the hypermedia specification, and overriding of methods is allowed only under strict circumstances to guarantee type consistency.

The composite schemas are defined using an enhanced version of a Z schema. The main distinction is that the schema can have a sequence of parameters. These parameters are used to instantiate the composite.

Finally, the navigation schema is presented with a collection of Abstract Design Perspective schemas (ADPs). ADPs are based on Abstract Design Views (ADVs) [CL95] and resemble the form of object-oriented VDM schemas [Ier91]. ADPs predicates, however, are specified using Z.

3.3 A conceptual model specification

The conceptual schema consists of three types of declarations: given type definitions, object definitions and axiomatic definitions. We will first define the notion of subtype in the scope of Hadez.

3.3.1 Subtyping

In order to introduce class constructors we first need to establish the notion of type equivalence and subtyping in the scope of Hadez.

3.1 Definition (Type equivalence) *Two types t_1 and t_2 are equivalent (denoted by $t_1 \equiv t_2$) iff $domain(t_1) = domain(t_2)$.*

In other words, two types are equivalent if their domains are the same, i.e. if they describe the same sets. This is known as structural equivalence of types [AC96].

3.2 Definition (Type of a value) *A value v is of type t, denoted by $v : t$, $\Leftrightarrow v \in domain(t)$*

Then we can define subtyping as:

3.3 Definition (Subtype) *A type t_i is a subtype of t_j (denoted by $t_i \sqsubseteq t_j$) iff $domain(t_i) \subseteq domain(t_j)$.*

The following lemma states that, if t_i is a subtype of t_j, then a value of type t_i is also a value of type t_j.

3.4 Lemma *$v : t_i$ and $t_i \sqsubseteq t_j \Rightarrow v : t_j$.*

Proof: Assume $a : t_i$ and $t_i \sqsubseteq t_j$. Since $domain(t_i) \subseteq domain(t_j)$, a has type t_j. □

We now turn our attention to the type constructors and how the subtyping relation applies to them.

3.5 Lemma *$t_1 \times t_2 \sqsubseteq t_a \times t_b$ if $t_1 \sqsubseteq t_a$ and $t_2 \sqsubseteq t_b$.*

Proof: We prove it by way of contradiction. Assume that there exists a value $\langle a, b \rangle$ in $t_1 \times t_2$ but not in $t_a \times t_b$. In order for this to be true, either a or b (or both) should not be elements of $domain(t_a)$ or $domain(t_b)$ respectively, which is a contradiction. □

3.6 Lemma *$t_1 \rightarrow t_2 \sqsubseteq t_a \rightarrow t_b$ if $t_a \sqsubseteq t_1$ and $t_2 \sqsubseteq t_b$.*

Proof: Assume there exists a function $f : t_1 \rightarrow t_2$. If $t_2 \sqsubseteq t_b$, then f returns, by definition, elements of type t_b. On the other hand, f will be undefined for any values outside the domain of t_1. The function f will be able to handle only parameters of type $t_a \sqsubseteq t_1$. Therefore, if $t_2 \sqsubseteq t_b$ and $t_a \sqsubseteq t_1$ then $f : t_a \rightarrow t_b$. □

Subtyping of classes

In the scope of Hadez, only the type signatures of classes are considered. We use the notation $C.l$ to refer to the field l of the class C.

Because classes should be properly defined, the inheritance relation (that is, the $C_p \prec C_c$ where C_p is an ancestor of C_c) creates a tree.

3.7 Definition (Type signature of a class) *Given a class* $C = \{P, l_i : t_i, i \in 1..n\}$*, its type signature, denoted by* $\tau(C)$ *is defined as*

$$\tau(C) = \begin{cases} \{l_i : t_i, i \in 1..n\} & \textit{if C has no parent} \\ \tau(P) \oplus \{l_i : t_i, i \in 1..n\} & \textit{otherwise} \end{cases}$$

In other words, the type signature of a class C is the type signature of its defined fields plus the type signature of its parent without the fields which were redefined in C.

3.8 Definition (Field override) *A field* $l : t_c$ *in a class C can override a field* $l : t_p$ *in class P,* $P \prec C$ *if* $t_c \sqsubseteq t_p$*.*

A field of an ancestor can be overridden only if the new type of the field is a subtype of the definition of that field in the ancestor class.

3.9 Lemma (Class subtyping) *A class C is a subtype of P,* $C \sqsubseteq P$ *if* $P \prec C$*.*

Proof: This result follows from definition 3.8. □

The notion of subtyping is important in Hadez. Its basic premise is that, under certain circumstances, it is useful to use an instance of an object of class C as one of P, given that $P \prec C$. This feature is particularly useful to take advantage of specialization: a specialized object can be used as an object of the type of its ancestor.

3.3.2 Given type definitions

Given types are user defined types which are considered atomic by the specification. The details of these types are not known. Given types are defined as their counterparts in Z. For example, the given types XML, IMAGE, DATE, and TECHNIQUE are defined with the following statement:

$[XML, IMAGE, DATE, TECHNIQUE]$

Each identifier in Hadez should be unique and consists of any combination of alphabetic characters, numbers and underscores, always starting with an alphabetic character.

The domain defined by each given type is incomparable with any other domain defined by another type.

3.3.3 Type constructors

Type constructors define new types from already-defined types.

Cross products and functions

Both cross products and functions are specified with Z abbreviation definitions. The following declaration defines a BIRTHDATE a total function that map from NAME to DATE:

$$BIRTHDATE == NAME \rightarrow DATE$$

As it was mentioned before, every identifier used in the definition of another identifier should be already defined. Therefore, it is expected that before this declaration both *NAME* and *DATE* are already defined.

Strictly speaking, a function is a subtype of a cross product. However, a function has special characteristics (as defined in section 2.2.4): for each element in their domain there is, at most, one element in their range.

Similarly, Z defines several types of cross products and functions, as defined in table 2.1 (on page 23):

These functions are useful for the specification of hypermedia applications, because they model relations, which will eventually become hyperlinks.

For instance, assume that we have a mapping between *artists* and *paintings*. The following definitions are significantly different, from the conceptual point of view:

$$PaintedByType_1 == painting \nrightarrow artist$$
$$PaintedByType_2 == painting \rightarrowtail artist$$

The type $PaintedByType_1$ specifies the set of partial functions in which, for each painting, there might not necessarily exist an artist; on the other hand, $PaintedByType_2$ specifies the set of functions in which for each painting there exists a different artist. In other words, if a function is defined as $PaintedBy : PaintedByType_1$, then not all paintings might have an artist (the function might be undefined for some paintings); if it is defined as $PaintedBy : PaintedByType_2$ then each painting in the collection is artisted by a different artist.

The use of these different types of functions will become more evident in the definition of composites within the structural schema.

Class constructor

The definition of classes of Hadez is similar to Z++. Hadez, however, is not concerned with the details of the implementation of the class. As a consequence, there is no distinction between methods and attributes. Hadez is concerned only with the data signature of a class.

Classes schemas are defined with an object schema:

ClassName[: *parentclass*]
$attr_1 : type_1$
...
$attr_n : type_n$

Where *ClassName* is the name of the class, *parentclass* is its parent and $attr_i$ is a field of type $type_i$. A class schema is, therefore, only the description of the type signature of the class.

In order to simplify type verification in Hadez, a class C is able to override a field $f : t$ of class P ($P \prec C$) iff the type of f in P has exactly the same as type as f as defined within C (subtyping will be allowed in other types of definitions, such as composites and perspectives). Since the class descriptions in Hadez are descriptions of their type signature, the inclusion in Hadez of the ability to override a field is purely a conceptual feature: a class C that inherits from P and only overrides fields from P (that is, it does not define new fields) has the same type signature as its parent P.

Here we exemplify the declaration of classes, in the scope of the Museum application discussed in chapter 2. We proceed to declare a class artifact with

two subclasses: sculpture and painting.

Artifact

$Title : string$
$CreationDate : DATE$
$Value : integer$
$Technique : string$
$Images : \mathbb{P}\, image$

Painting : *Artifact*

$Dimensions : real \times real$

Sculpture : *Artifact*

$Weight : real$

The first class, *Artifact* does not have any ancestor. Both *Painting* and *Sculpture* are descendents of *Artifact*.

As it was described in section 2.2.4 (in page 22), only single inheritance is supported and the descendent is considered a subtype of the ancestor. Therefore, an instance $x : Painting$ is also an instance of *Artifact*, i.e. $x : Artifact$.

3.3.4 Instances

An instance is an object which resides in a hyperbase and whose type is defined in the conceptual schema. During the declaration of instances, the designer specifies what are the different sets of instances of each different type, any constants, and any relationships between them. These declarations are specified using Z axiomatic definitions.

For instance, in the Museum application, we need to declare the following sets: the set of artifacts in the museum, and the set of artists. Furthermore, we define *Anonymous* as an element of *Artists* which is to be used in the specification, as the author for those paintings whose author is unknown.

$$
\begin{array}{|l}
Collection : \mathbb{P}_1\, Artifact \\
Artists : \mathbb{P}_1\, Artist \\
Anonymous : Artist \\
\hline
Anonymous \in Artists \\
\end{array}
$$

Collection is a set of non-empty artifacts. *Artists* is a non-empty set of artists. *Anonymous* is defined as an instance of type *Artist*, which is an element of the set *Artists*. Because *Painting* and *Sculpture* are both subtypes of *Artifact*, *Collection* can consist of a mixture of objects of type *Painting*, *Sculpture* or *Artifact*.

3.3.5 Relations definitions

In [YB00], Yoo and Biever identified that "a vital aspect of hypermedia design is identifying relationships and implementing them as links" and that "many relationships are poorly identified or ignored in current hypermedia design methodologies". RMM stresses this fact by acknowledging that "hypermedia is a vehicle for managing relationships among information objects" [ISB95].

Methodologies such as Yoo's Relationship Navigation Analysis (RNA) [YB00] can identify the different relations that can be exploited in a hypermedia application. These relations can then be explicitly described in Hadez as instances of cross products and functions.

Relations are instances in the hyperbase. They are either functions or relations on object instances in the hyperbase and they are described—as object instances—with axiomatic definitions. The axiomatic definitions consist of two parts: its type signature and a sequence of predicates that bind (and potentially restrict) the values of the domain and range of the relation. For example, the relation *CreatedBy* binds artifacts and artists:

$$
\begin{array}{|l}
CreatedBy : Artifact \nrightarrow Artist \\
\hline
\mathrm{dom}\, CreatedBy = Collection \\
\mathrm{ran}\, CreatedBy = Artists \\
\end{array}
$$

In this example, *CreatedBy* is a partial function. Its domain is the entire collection, hence the function is defined for every artifact in the collection. Its range is defined to be the set of *Artists*, therefore, there are no artists in *Artists* that have not created an artifact in the collection. Finally, every painting has

only one creator (which might be *Anonymous*, which is an instance of type *Artist*).

The function *CreatedBy* is fundamental in the specification of the design of the museum. Both the type signature of the relation and the predicate section of the relation, convey a lot of information about the characteristics of the final application. By using a different type of function we can also alter the characteristics of this relationship.

For example, the previous declaration states that there is only one author for each artifact. We can easily change this declaration to indicate that there are zero or more authors for one artifact:

$$\begin{array}{|l} CreatedBy : Artifact \pfun \mathbb{P}\, Artist \\ \hline \mathrm{dom}\, CreatedBy = Collection \\ \mathrm{ran}\, CreatedBy = Artists \end{array}$$

We can strengthen this even further by replacing $\mathbb{P}$ for $\mathbb{P}_1$, a set of at least one element; hence an artifact should have at least one author.

In the following example, there is a one-to-one correspondence between artifacts and artists; in other words, each artifact is made by a different artist and for each artist there is one artifact in the database:

$$\begin{array}{|l} CreatedBy : Artifact \rightarrowtail\!\!\!\!\!\twoheadrightarrow Artist \\ \hline \mathrm{dom}\, CreatedBy = Collection \\ \mathrm{ran}\, CreatedBy = Artists \end{array}$$

The predicate section can also state many facts of the hyperbase. For example, the following declaration for *CreatedBy* indicates that there might be some artists that do not have a painting in the hyperbase:

$$\begin{array}{|l} CreatedBy : Artifact \pfun \mathbb{P}\, Artist \\ \hline \mathrm{dom}\, CreatedBy = Collection \\ \mathrm{ran}\, CreatedBy \subseteq \mathbb{P}\, Artists \end{array}$$

As a result of subtyping, *CreatedBy* is also defined for elements of type *Painting* and *Sculpture*.

3.4 Structural schema

The structural schema of an application is described with a collection of composite schemas. A composite schema specifies how a composite should be instantiated from other composites or from object instances in the hyperbase.

The parts of the definition of a composite are:

- A unique name identifying it.
- An instantiation parameter list. It is a sequence of objects which are used to instantiate the composite. This list can be empty.
- Ancestor. It specifies whether the composite is a descendent of another composite.
- Attributes Declaration. A sequence of attributes local to the composite. The atttributes can be either objects in the hyperbase, or other composites.
- Predicate section. It is a sequence of Z predicates that bind the value of the local attributes to the parameter of the composite.

The composites are defined with a composite schema. The general form of a composite schema is:

Ξ*CompositeName* (parameters list) : *ParentComposite*

$attr_1 : type_1$
...
$attr_n : type_n$

$predicate_1$
...
$predicate_n$

The composite name is decorated with Ξ to clearly differentiate it from other Hadez declarations. The parameter list is a sequence of labels, each with a corresponding type; this list can be potentially empty. The composite might be a child of another composite: *ParentComposite*. The attribute $attr_i$ has type $type_i$;

$predicate_i$ is a predicate on the attributes and parameters of the composite or on any global objects. The following schema is an example of a simple composite:

$$\Xi ArtifactComposite(Art : Artifact)$$
$$Author : Artist$$
$$CreatedBy(Art, Author)$$

The *ArtifactComposite* takes one parameter (*Art* of type *Artifact*). It declares one attribute, its author, which is bound to the creator of the artifact by the only predicate in the predicate section. This artifact illustrates the most important feature of composites: they are higher level entities which are created by exploiting the declared relations between instances in the hyperbase.

The parameter list is optional. This is illustrated by the following composite, which creates a composite of all the artifacts in a collection:

$$\Xi AllArtifacts$$
$$WholeCollection : \mathbb{P}\, ArtifactComposite$$
$$\forall P \in Collection \exists Art \in WholeCollection \bullet \Xi Art(P)$$

The predicate section of this composite uses a construction unique to Hadez and requires further explanation. A composite instantiation is the creation of a composite from a list of parameters. The statement ΞC(parms list) for a variable C of a composite type t, is equivalent to creating a composite of type t with parameters parms list and then assign the returned composite to the variable C.

In the previous composite, the following predicate creates a composite for each element P in *Collection*:

$$\forall P \in Collection \exists Art \in WholeCollection \bullet \Xi Art(P)$$

The same statement can be written in a more succinct way by defining a set iteration operator:

$$[X]$$
$$\Xi \_ \circ \_ : composite(X) \times \mathbb{P}\, X \rightarrow \mathbb{P}\, composite(X)$$
$$\forall\, C : composite(X);\ S : \mathbb{P}\, X \bullet$$
$$\quad \Xi C \circ S = \{x : composite(X) \mid (\exists y : S \bullet x = \Xi C(y))\}$$

In other words, the set iteration operator apples a composite constructor to each of the elements of the parameter set and returns the set of composites. We can then rewrite the previous predicate as:

$$WholeCollection = \Xi ArtifactComposite \circ Collection$$

In this case, a composite constructor (*ArtifactComposite)* is iterated over each element in *Collection*; each composite constructor takes one element in *Collection*. The resulting (*WholeCollection*) is a set of artifact composites with the same number of elements as *Collection*.

3.4.1 Creating indices, guided tours and navigational contexts

Each of the three main hypermedia design methodologies (HDM, RMM and OOHDM) recognizes the importance of creating groups of "entities" which can be navigated as a group. RMM and HDM use the term "guided tour", while OOHDM uses the "navigational context". In both cases, they refer to groups of similar objects, defined either by enumeration (listing each one of its members) or by a predicate (list all the objects that fulfill the following property). The methodologies further define design abstractions that allow the reader to navigate these groups of entities.

Hadez takes the position that both guided tours and navigational contexts are special cases of composites. In these cases, the composites are sets, sequences, or bags of other composites which are to be shown to the reader as a group. We illustrate this idea with the following composite:

$$\Xi ArtistComposite(ArtistName : string)$$
$$Person : Artist$$
$$Works : \mathbb{P}\, ArtifactComposite$$
$$Person.Name = ArtistName$$
$$Works = \Xi ArtifactComposite \circ CreatedBy(\!|\ \{Person\}\ |\!)$$

ArtistComposite is instantiated with the name of the artists (*Name*) and creates a composite with the corresponding artist and her works. The first predicate of the composite binds the value of the parameter *Name* with an object of type *Artists* whose attribute *Name* is the same. Note that there is an assumption (not stated in the specification) that all the names of the artists are unique. This composite can be used to define a guided tour (a navigational context) of all the artifacts of a given painter.

It it also possible to define composites in which their elements share one common feature. In the following schema, the composite *TourTechnique* is intended to be a group of artifacts that are created using the same technique:

$$
\begin{array}{|l}
\hline
\Xi TourTechnique(Technique : TECHNIQUE) \\
Artifacts : \mathbb{P}\, ArtifactComposite \\
\hline
\forall a \in Collection \bullet \\
\quad a.Technique = Technique \; \exists Art : Artifacts \bullet \\
\quad\quad \Xi ArtifactComposite(a) \\
\hline
\end{array}
$$

We could then instantiate the tour of surrealist paintings (assume that *surrealism* is a constant of type TECHNIQUE that corresponds to the surrealism technique) with the following declaration:

$$
\begin{array}{|l}
\hline
\Xi SurrealismTour \\
Tour : TourTechnique \\
\hline
\Xi Tour(surrealism) \\
\hline
\end{array}
$$

SurrealismTour could then be presented as a guided tour in a variety of ways: a sequence of hyperpages each showing one of the artifacts; or by creating a table of contents hyperpages that links to each one of the different paintings. The perspective schema will specify how the composite is presented to the user.

3.4.2 Specializing composites

As it was described above, a composite schema can inherit its characteristics from another one. The semantics of inheritance require further explanation. A composite C can be inherited from composite P if C is type compatible with P.

3.10 Definition (Type Compatible Composite) *A composite C_c with parameters list $(p^c_1 : t^c_1, ..., c^c_i : t^c_i)$ is said to be type compatible with composite C_p with parameters list $(p^p_1 : t^p_1, ..., c^p_j : t^p_j)$ iff $i \geq j$ and $\forall k \in 1...j\, p^c_k = p^p_k$ and $t^c_k \sqsubseteq t^p_k$.*

In other words, a composite C is type compatible with composite P if C has at least the same parameter names in the same order as in P, and for each common parameter, the type of the parameter of C is a subtype of the parameter of P.

3.11 Axiom *A composite C can inherit from a composite P if C is type compatible with P.*

This restriction is necessary in order to guarantee the type verification of the composite: As long as a parameter of the child is a subtype of the corresponding parameter in the parent, the predicates of the parent are still defined in the child.

Assume the following two composites *Parent*, and *Child*, which inherits from *Parent*. By definition, *Child* is type compatible with *Parent*.

$\Xi Parent(p^p_1 ... p^p_i)$

$attr_1 : type_1$
...
$attr_m : type_n$

$predicate_1$
...
$predicate_k$

$\Xi Child(p^c_1 ... p^c_j) : Parent$

$attr_{m+1} : type_{m+1}$
...
$attr_{m+n} : type_{m+n}$

$predicate_{k+1}$
...
$predicate_{k+l}$

Child is equivalent to the composite *Equivalent*:

$\Xi Equivalent(p^c_1 ... p^c_j)$

$attr_1 : type_1$
...
$attr_m : type_n$
$attr_{m+1} : type_{m+1}$
...
$attr_{m+n} : type_{m+n}$

$predicate_1$
...
$predicate_k$
$predicate_{k+1}$
...
$predicate_{k+l}$

By inheriting from a composite, the new composite inherits all its attributes and predicates. Inheritance is a syntactical construct which is intended to modularize the specification of composites.

The simplest child of a composite will not define new attributes or predicates and is used only to strength the type signature of the parameters of the composite. The following composite, *PaintingComposite*, is a specialization of *ArtifactComposite* which can be instantiated only with a parameter of type *Painting*.

$\Xi PaintingComposite(Art : Painting) : ArtifactComposite$

In the following example we strengthen the predicate section of the previous composite by forcing that the author cannot be *Anonymous*.

$\Xi PaintingCompositeNotAnon(Art : Painting) : PaintingComposite$

$Author \neq Anonymous$

It is also possible to add attributes in the child composite. The following composite enhances *ArtistComposite* by adding a counter of the number of paintings by that artist.

$$\Xi ArtistCompositeExtended : ArtistComposite$$
$$NumberOfWorks : int$$
$$NumberOfWorks = \#CreatedBy(\!| \{Person\} |\!)$$

3.4.3 Generic composite schemas

Some times, several composites share almost the same declaration, except that they apply to different types. In such cases, it is desirable to define a composite without specifying the type to which the composite applies. This concept is similar to the declaration of abstract data types—such as stacks and lists—in which the properties of the ADTs are defined precisely in terms of a generic type.

In Hadez, a generic composite schema is a composite schema that uses generic types. A generic type is just a label for a type. When the generic composite schema is instantiated into a composite schema, this type should be specified. The form of a generic composite schema is:

$$\Xi GenericCompositeName[\text{generic types list}](\text{parm list})$$
$$attr_1 : type_1$$
$$\ldots$$
$$attr_m : type_n$$
$$predicate_1$$
$$\ldots$$
$$predicate_k$$

And a generic composite is instantiated with a generic composite instantiation:

$$Composite == GenericCompositeName[\text{sequence of types}]$$

This generic schema declaration would be equivalent to the following declaration, in which any reference to a generic type is replaced by the corresponding type, according to the genetic composite instantiation statement.

$$\Xi CompositeName[\text{generic types list}](\text{parm list})$$
$$attr_1 : type_1$$
$$\ldots$$
$$attr_m : type_n$$

$$predicate_1$$
$$\ldots$$
$$predicate_k$$

The following generic composite *Tour* takes a generic type parameter *X*. The generic composite has two parameters: a set of elements, and a function that compares two elements. The objective of the generic composite is to create tours of sequences of elements ordered according to the parameter function *LessThan*.

$$\Xi Tour[X](Elements : \mathbb{P}\, X, LessThan : X \times X \rightarrow boolean)$$
$$Sequence : \operatorname{seq} X$$

$$\operatorname{ran} Sequence = Elements$$
$$\forall\, i \in 2..\#Sequence \bullet$$
$$\quad LessThan(Sequence(i-1), Sequence(i))$$

We can then instantiate a composite schema by specifying the type that the generic type parameter takes. In the following declaration, *ArtistComposite* replaces X in the generic composite to create a new composite schema *OrderedArtistTour*:

$$OrderedArtistTour == Tour[ArtistComposite]$$

This composite schema is equivalent to:

$$\Xi OrderedArtistTour(Elements : \mathbb{P}\, ArtistComposite,$$
$$\qquad LessThan : ArtistComposite \times ArtistComposite \rightarrow boolean)$$
$$Sequence : \operatorname{seq} ArtistComposite$$

$$\operatorname{ran} Sequence = Elements$$
$$\forall\, i \in 2..\#Sequence \bullet$$
$$\quad LessThan(Sequence(i-1), Sequence(i))$$

We can also define an ordered tour of *Paintings*:

OrderedPaintingsTour == *Tour*[*PaintingComposite*]

The main goal of generic composite definitions is to promote reuse.

3.4.4 Grammar

Figure 3.1 shows an abbreviated description of the grammar of Hadez corresponding to the definition of composites.

Composite ::= *CSchema*
| *CGenSchema*
| *CGenSchemaIns*

CSchema ::= *[* ΞCName *[* **(** *ParmList* **)** *][* : CName *]*
DeclList

AxiomPart *]*

CGenSchema ::= *[* ΞCName **[** *GenParms* **]** *[* **(***ParmList***)** *]*
DeclList

AxiomPart *]*

CGenSchemaIns ::= CName==CName**[***GenParmsIns***]**

DeclPart ::= *BasicDecl* ... *BasicDecl*

BasicDecl ::= Name: *(*,Name: *)*∗ : TypeName

AxiomPart ::= *(Predicate)*+

ParmList ::= *BasicDecl (*, *BasicDecl)*∗

GenParmList ::= GenType *(*,GenType*)*∗

GenParmIns ::= TypeName*(*,TypeName*)*∗

Figure 3.1: Grammar for composites.

3.5 Perspective schema

The specification of a hyperbase indicates the characteristics of the domain specific data of an application and its composite schemas. Composites, however, are not displayed directly to the reader. The reader perceives and interacts with composites through perspectives.

In HDM, the term *perspective* is used to refer to different ways in which the same composites can be presented to the reader and states that the specification of the perspectives is outside the scope of the methodology [GPS93]. In OOHDM, the concept of perspective is served by Abstract Design Views (ADV). ADVs are software engineering design artifacts intended to separate the user-interface from application objects. They were first defined in software engineering [CILS93, CL95] and later adapted to OOHDM [RSLC95, Ros96]. In OOHDM, "an ADV describes, in an abstract, implementation independent way, a number of relationships including the media objects perceived by the user of the hypermedia application, the mode of interaction with these objects, and the interface transformations that occur while navigating through the hypermedia" [RSLC95]. ADVs, contrary to HDM's perspectives, not only specify how the object is presented to the reader; they also specify how the interaction between the user and the observed object should proceed.

The software engineering ADVs are specified using schemas in a notation based on the formal specification language VDM [CILS93]. OOHDM, however, uses a graphical notation to describe ADVs based on JASMINUM [CC94]. Figure 3.2 is an example of an ADV description in OOHDM. The diagram shows two ADVs: *Art Gallery* and *Painting*, one defined to contain the other. The ADVs indicate which fields of the composites are shown and their location in the viewport. The description of the reader interaction is specified in an ADVChart. Figure 3.3 shows the ADVChart for the ADV Painting. It indicates what the ADV should do in response to the reader's actions.

As HDM and OOHDM do, Hadez separates the specification of application domain objects from how they are perceived by the reader. The same object can be presented in a variety of forms, each particularly suited for the objective of the application or for a particular type of reader. In order to accomplish this task, Hadez uses the notion of Abstract Design Perspective or ADP.

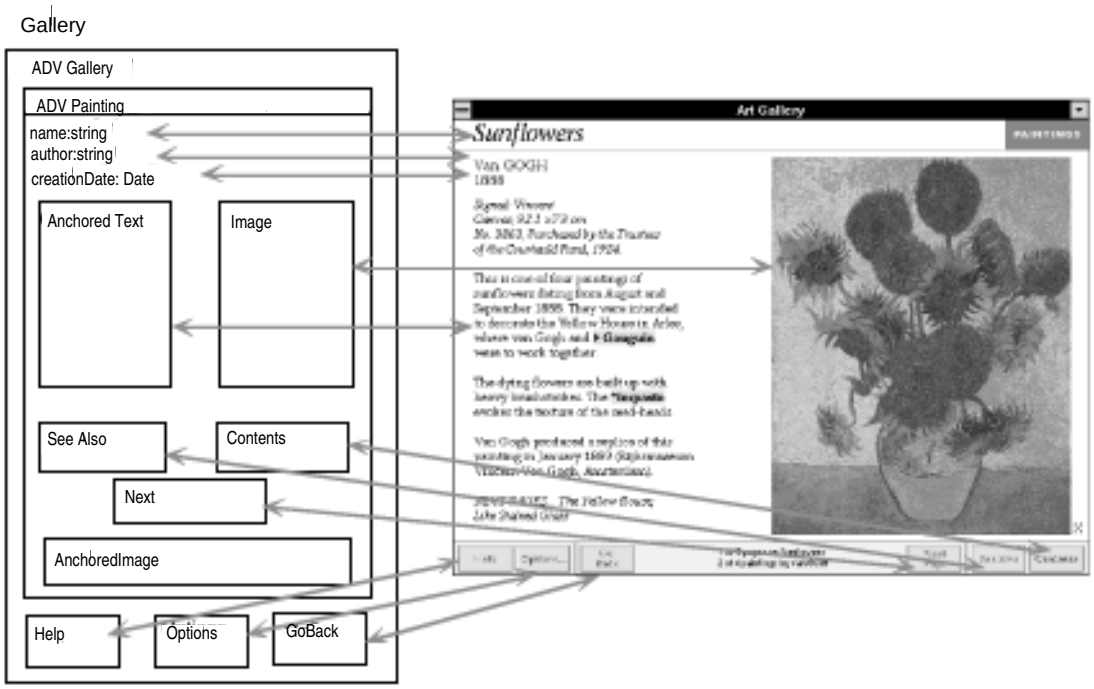
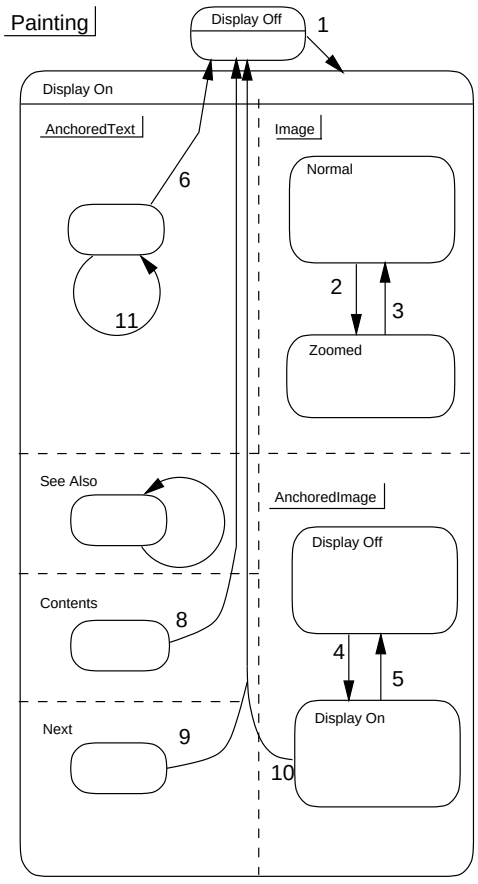


Figure 3.2: Graphical representation of an ADV for a painting. Figure from [RSLC95].



Transitions

1:
Event: Display
Pre-Cond:
Post-Cond: perceptionContext =perceptionContext + Painting

2:
Event: MouseClicked
Pre-Cond: focus (Image)
Post-Cond: self.DisplayZoomed

3:
Event: MouseClicked
Pre-Cond: focus (Image)
Post-Cond: self.DisplayNormal

4:
Event: ShowAnchors
Pre-Cond:
Post-Cond: perceptionContext = perceptionContext + AnchoredImage

5:
Event: MouseClicked
Pre-Cond: not (focus (AnchoredImage))
Post-Cond: perceptionContext = perceptionContext - AnchoredImage

6:
Event: MouseClicked
Pre-Cond: focus (Anchors (i))
Post-Cond: owner.anchorSelected (Anchors (i)) ; perceptionContext = perceptionContext - Painting

7:
Event: MouseClicked
Pre-Cond: focus (SeeAlso)
Post-Cond: ShowAnchors

8:
Event:: MouseClicked
Pre-Cond: focus (Contents)
Post-Cond: owner.anchorSelected (Contents); perceptionContext = perceptionContext - Painting

9:
Event:: MouseClicked
Pre-Cond: focus (Next)
Post-Cond: owner.anchorSelected (Next); perceptionContext = perceptionContext - Painting

10:
Event: MouseClicked
Pre-Cond: focus (Anchors(i))
Post-Cond: owner.anchorSelected (Anchors (i)); perceptionContext = perceptionContext - Painting

Figure 3.3: User interaction for the ADV *Painting*. Figure from [RSLC95].

Like an ADV, an ADP describes how a reader should perceive a given object and how the interaction with the user alters this perception. It specifies also what becomes an anchor of a hyperlink and what it means to select such an anchor. An ADP is based on ADVs and can be considered their descendent. ADPs further improve ADVs by providing specific features that are particularly suited for hypermedia design.

An ADP observes a composite and one composite can be observed by one or more ADPs at the same time.

In Hadez, an ADP models a perspective. Like a perspective, the main characteristics of ADPs are:

- An ADP has a collection of local attributes that define a set of states. This set of states can be infinite.
- An ADP can change its state only through input messages. Input messages are generated by the reader, by the run-time system or by other regions within the current viewport. A change from one state of the region to another is known as a transition.
- A transition can trigger an output message that is intended for ADPs in other regions.
- The ADP specifies, for each state, what parts of the observed composite are shown to the reader. The set of attributes shown to a reader in a given state is known as the *current selection* of the ADP. A transition of the ADP might determine a change in the current selection. The ADP, therefore, has the ability to present different information to a reader at different times.
- An ADP can consist of a finite number of ADPs. The communication between the different component ADPs is done through internal messages.

The viewport of a runtime system is divided into regions, as described in section 2.5.4. Each of these regions is modeled with one ADP. The result is that the viewport corresponds to an ADP, potentially consisting of one or more ADPs.

The main goals of ADPs are to be able to specify the complex interaction between the hyperbase and the reader and to promote the reuse of this specification.

3.5.1 ADP schemas

ADP Name [: Ancestor] **Observes** Composite:Type
Declaration: $attrname_1 : type_1, ..., attrname_n : type_n$,...,
Block $block_1$**:** *block specification*
Block $block_2$**:** *block specification*
...
Preconditions: *predicates*
Invariants: *predicates*
Message $event_1$**:** $post-condition$ *predicate*
Message $event_2$**:** $post-condition$ *predicate*
...
End Name

Figure 3.4: An Abstract Design Perspective specification schema.

Figure 3.4 shows the general form of an ADP schema. Each ADP has a unique name (*Name*). The ADP can, optionally, inherit its characteristics from another ADP (*Ancestor*). An ADP observes always a composite named *Composite* with type *Type*. The state of the ADP is determined by the value of the local variables defined in its declaration section. The predicates in the preconditions section are constrains that bind—an therefore initialize—the values of its local variables. The current state of the perspective states indicates which attributes (of its observed composite) the perspective shows at a given time. It is important to mention that the perspective can only show to the reader attributes of the composite. It cannot show the values of its internal variables.

The preconditions of the perspective determine the initial state of the ADP. The invariants section is a sequence of predicates that should be true at any state of the ADP. Blocks define regions within the current ADP which can be considered independent from other blocks within the current ADP. In its simpler form, a block is a group of attributes of the observed composite. Finally, every ADP is able to handle a given set of messages, which are originated outside the ADP or internally. In the following sections, we will introduce ADPs through

examples. The characteristics and properties of perspective will be formalized in section 5.4 (on page 120).

3.5.2 A simple ADP schema

A very simple ADP is depicted in figure 3.5. This ADP observes a composite of type *CompositeType* and always displays its attributes $attr_1$, $attr_2$ and $attr_3$. The result of the predicate $show(attr)$ for any attribute *attr* is always true and its side effect is to make *attr* visible. Its counterpart, $hide(attr)$, is always true, and has the opposite effect than *show*, and makes *attr* invisible.

ADP ADP_1 **Observes** $Composite : Type$
Invariants:
$show(Composite.attr_1)$
$show(Composite.attr_2)$
$show(Composite.attr_3)$

End ADP_1

Figure 3.5: A simple ADP which always displays three attributes of a composite.

3.5.3 Messages

The preceding ADP can be enhanced to respond to a message *toggle_attr* which will toggle the visibility of $attr_1, attr_2$, and $attr_3$. It is assumed that *toggle_attr* is going to be generated by another ADP and, therefore, it is an external message. The enhanced ADP is depicted in Figure 3.6. ADP_2 has a local variable *Visible* which is used to keep track of whether the attributes are visible or not. The *preconditions* force the three attributes to be displayed at the starting state. In the message section for *toggle_attr*, depending on the value of *Visible* the attributes are then shown or hidden. A variable name followed by an denotes the value of the variable after the message section. A message section cannot modify the value of the attributes of the composite it observes, only the value of its internal variables.

ADP ADP_2 **Observes** *Composite* : *Type*
Declarations:
Visible: boolean
toggle_att: **input** message
Preconditions:
$show(Composite.attr_1)$
$show(Composite.attr_2)$
$show(Composite.attr_3)$
$Visible = TRUE$
Message *toggle_att*:
$Visible \Rightarrow$
$hide(Composite.attr_1)$
$hide(Composite.attr_2)$
$hide(Composite.attr_3)$
$\neg Visible \Rightarrow$
$show(Composite.attr_1)$
$show(Composite.attr_2)$
$show(Composite.attr_3)$
$Visible' = \neg Visible$
End ADP_2

Figure 3.6: An ADP depicting the use of messages.

3.5.4 Blocks

The most conspicuous characteristic of ADP_2 is the repetition of the predicates to show the three attributes. In order to avoid unnecessary repetition, ADPs use the concept of blocks. A block is an independent region of the current composite. A block can consist of:

- A finite set of attributes of the current composite.
- An ADP that observes a composite which is part of the current composite.

The second form of a block will be described later in this chapter. In the first form, a block becomes an abbreviation to refer to a group of attributes. The action of showing the block results in showing all the components of the block, and hiding the block has the opposite result. As a consequence, the visibility of a group of attributes can be toggled with just one predicate. By default, a block is visible at its starting state. Figure 3.7 shows ADP_{2b}, which defines a block, consist of three attributes and which has the same functionality as ADP_2.

ADP ADP_{2b} **Observes** *Composite* : *Type*
Declarations:
Visible: boolean
toggle_att: input message
Block $block_1$**:**
$Composite.attr_1$
$Composite.attr_2$
$Composite.attr_3$
Preconditions:
$Visible = TRUE$
Message *toggle_att*:
$Visible \Rightarrow hide(block_1)$
$\neg Visible \Rightarrow show(block_1)$
$Visible' = \neg Visible$
End ADP_{2b}

Figure 3.7: This ADP exemplifies the use of blocks.

3.6 Buttons

Buttons are a user interface mechanism which allow the reader to generate input messages to the current ADP. A button has two attributes: its label, which usually is an attribute of the observed composite; and the message that it generates. A button, like a hyperlink, can be selected by the reader; when a button is selected, its corresponding message is sent to the current ADP. The main objective of the buttons is to allow the reader to change the current state of the ADPs currently displayed. As a result, the ADPs might display, in the new state, different attributes of their currently observed composites.

Figure 3.8 shows an ADP which extends the functionality of ADP_{2b} (declared in figure 3.7) with a button that, when selected, sends the message *toggle_att* to the current ADP. In this example the label of the button is $attr_4$; that is, the data shown to the reader to be selected, similar to an anchor of a hyperlink.

3.6.1 Pagination

As it was described in section 2.5.1, a composite can be broken into more than one perspective, where each perspective is presented in different hyperpages. In order to specify pagination, it is necessary to indicate:

ADP ADP_3 **Observes** *Composite* : *Type*
Declarations:
Visible: boolean
toggle_att: internal message
Block $block_1$**:**
$Composite.attr_1$
$Composite.attr_2$
$Composite.attr_3$
Block $block_2$**:**
$Composite.attr_4$ **button** *toggle_att*
Preconditions:
$Visible = TRUE$
Message *toggle_att*:
$Visible \Rightarrow hide(block_1)$
$\neg Visible \Rightarrow show(block_1)$
$Visible' = \neg Visible$
End ADP_3

Figure 3.8: An ADP with a button that generates an internal message.

- What attributes of the observed composite of the current perspective are to be presented in other perspectives.
- How is the current perspective going to link to the other perspectives.

ADP_4, in figure 3.9 exemplifies this feature. The attributes $attr_3$ and $attr_4$ are presented in a different hyperpage. They are presented with two different ADPs: $OtherADP_1$, $OtherADP_2$, respectively. A link requires an anchor (the data that is displayed to the reader in the starting point of the hyperlink). In this particular example, the anchors are the attribute x of $Composite.attr_2$ and the attribute y of $Composite.attr_3$, respectively. These links represent structural links. In this particular example, $OtherADP_1$ and $OtherADP_2$ denote other hyperpages within the application.

3.7 Building complex ADPs

ADPs can be combined into more complex ADPs. There are three ways in which ADPs can be combined:

ADP ADP_4 **Observes** *Composite* : *Type*
Block $block_1$**:**
anchor $Composite.attr_3.x$ **linkto** $OtherADP_1(Composite.attr_3)$
anchor $Composite.attr_4.y$ **linkto** $OtherADP_2(Composite.attr_4)$
End ADP_4

Figure 3.9: This ADP demonstrates the use of structural links and splits the composite in multiple ADPs.

- Aggregation. One or more blocks in an ADP are described with other ADPs. These ADPs are considered part of the current ADP.
- Inheritance. One ADP is extended into a more specialized one by way of inheritance.
- Parallel composition. An ADP is the result of composing two or more ADPs.

3.7.1 Aggregation

One of the most important characteristics of a region is that it can be composed by other regions and the behavior of the component regions determine the behavior of the region that compose them. Hadez takes advantage of this feature: ADPs can be aggregated and become "components" of another ADP.

This is done by declaring an ADP a block of the current ADP. These blocks, as any other, can be instructed to be hidden or shown. The difference between an ADP block and a regular block is, first, that the ADP block can have an "extended" state as defined by the attributes of its ADP; second, an ADP block can handle input messages sent by other ADP blocks or by the current ADP; and third, it can generate messages intended for other ADPs.

Aggregation has the following characteristics:

- The current ADP does not have access to the internals of its aggregated ADPs. The current ADP can communicate only with its aggregated ADPs by sending them messages; and vice-versa, the aggregated ADPs can communicate only with the aggregating ADP or other aggregated ADPs by using messages.

- The only operations allowed by an ADP on its aggregated ADPs are "show" and "hide". In other words, the current ADP has the ability to operate on the ADP block only as if it was a block. When the aggregated ADP is "shown", what the ADP actually displays is determined by the aggregated ADP's own state, like any other ADP.
- The composite that each aggregated ADP observes is an attribute of the composite that the aggregating ADP observes. One important feature of aggregated ADPs is that they can observe different composites at different times, with the following restrictions:
  - An aggregated ADP can observe only composites of a given type.
  - An ADP is said to be *attached* to a given composite when the ADP is ordered to observe such composite. Composites can be statically or dynamically attached to an ADP. A composite is said to be statically attached to an ADP if the ADP is ordered to observe the aforementioned composite as a preconditions in the aggregating ADP. Otherwise it is said to be dynamically attached. Under static aggregation the aggregated ADP only observes one composite at all times; while under dynamic aggregation the ADP can observe different composites at different times.
  - The aggregating ADP is responsible for attaching composites to the aggregated ADPs.
  - An ADP should always observe a composite.
- The output messages of the aggregated ADPs are always seen by the current ADP. The other ADPs do not, however, see the output messages of other ADPs. The only way to send a message from one aggregated ADP to another is by creating a message handler in the aggregating ADP that triggers a second message to the destination ADP block when the source ADP generates the first message.
- Any output messages of the aggregated ADPs which are not handled by the aggregating ADP become part of the output messages of the latter.
- The input messages of the aggregated ADPs become part of the input messages that the aggregating ADP can handle.

ADP ADP_5 **Observes** *CompositeType* : *Type*
    Block $block_1$**:**
        ADP ADP_i **Observes** $Composite.attr_1$
    Block $block_2$**:**
        ADP ADP_j
    Message $mess_1$**:**
        attach $attr_2$ **to** $block_2$
End ADP_5

Figure 3.10: An example of aggregation.

We illustrate the use of aggregation with ADP_5, defined in figure 3.10. ADP_5 has two blocks. The block $block_1$ corresponds to ADP_i, which statically observes $attr_1$. In the case of $block_2$, ADP_j is attached to $attr_2$ inside the message section $mess_1$.

Figure 3.11 defines ADP_6; this ADP has the same functionality as ADP_3 (defined in figure 3.8 on page 70). The message that corresponds to the button defined in $block_2$ sends *toggle_att* to the block $block_1$.

ADP ADP_6 **Observes** *Composite* : *Type*
    Block $block_1$**:**
        ADP ADP_{2b} **Observes** Composite
    Block $block_2$**:**
        $Composite.attr_4$ **button** $block_1$*.toggle_att*
End ADP_6

Figure 3.11: ADP_{2b} is composed into a more complex ADP_5, which has the same functionality as ADP_3.

Aggregation can be used to model complex regions, by dividing its specification in less complex ADPs that, when combined, reflect the desired region. Figure 3.12 shows a region that is composed of three regions, one of which is composed of two regions. In order to illustrate how composites can be aggregated, we will model this region by using one ADP for each region.

The resulting ADP schemas are shown in figure 3.13. This shows how ADPs can be embedded at multiple levels and how, when they are presented to the reader, they occupy a single region in the viewport of the run-time system.

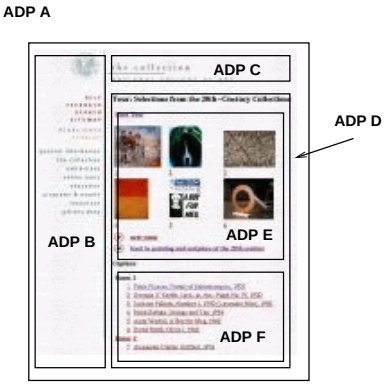


Figure 3.12: A more complex region.

3.7.2 Inheritance

Inheritance is a mechanism to extend an existing ADP with new features. A composite ADP_c (which observes a composite C_c of type t_c) can be a child of ADP_p (which observes a composite C_p of type t_p if $t_c \sqsubseteq t_p$).

The properties of a child ADP are:

- The restriction $t_c \sqsubseteq t_p$ is necessary in order to guarantee that every predicate in the parent is also valid in the child ADP.
- All the declarations of the parent are available to the child.
- The preconditions (and the invariants) of the child are the conjunction of the preconditions (and the invariants) of the parent and the preconditions (and the invariants) as defined by the child.

A child ADP can override a message handler and a block definition. This feature provides basic polymorphism to ADPs. For instance, let's assume that the parent has a block with one attribute $attr_1$. A child can redefine the block by including $attr_2$ instead. When a predicate in the parent (inherited by the child) shows this block, the block defined in the child is the one actually shown. The same applies for message handlers: a message can be sent by a predicate in the parent and handled by the message handler as redefined by the child.

Figure 3.14 shows an example of inheritance. ADP_7 is a child of ADP_{2b} (defined in figure 3.7 in page 70). In this case, no part of the parent is redefined;

ADP *ADP_B* **Observes** $Composite_b : Type_b$
Invariants:
...
End *ADP_B*

ADP *ADP_C* **Observes** $Composite_c : Type_c$
Invariants:
...
End *ADP_C*

ADP *ADP_E* **Observes** $Composite_e : Type_e$
Invariants:
...
End *ADP_E*

ADP *ADP_F* **Observes** $Composite_f : Type_f$
BLock *block_1*:
...
End *ADP_F*

ADP *ADP_D* **Observes** $Composite_d : Type_d$
BLock *block_1*:
ADP ADP_E **Observes** $Composite_d.attr_1$
BLock *block_2*:
ADP ADP_F **Observes** $Composite_d.attr_2$
End *ADP_D*

ADP *ADP_A* **Observes** $Composite : Type$
BLock *block_1*:
ADP ADP_B **Observes** $Composite.attr_1$
BLock *block_2*:
ADP ADP_C **Observes** $Composite.attr_2$
BLock *block_3*:
ADP ADP_D **Observes** $Composite.attr_3$
End *ADP_A*

Figure 3.13: A collection of ADPs that model the region depicted in figure 3.12.

hence, the resulting ADP inherits all the declarations of its parent. The message generated by its button in $block_2$ is handled by $block_1$ defined in the parent ADP.

ADP $ADP_7 : ADP_{2b}$ **Observes** *Composite* : *Type*
    Block $block_2$**:**
        $Composite.attr_4$ **button** *toggle_attr*
    Preconditions:
        $show(block_2)$

End ADP_7

Figure 3.14: An ADP which inherits its characteristics from another.

The schema in figure 3.15 shows the definition of a polymorphic ADP. ADP_8 inherits from ADP_7 which itself inherits from ADP_{2b}. ADP_8 redefines $block_1$ (originally defined in ADP_{2b}). When the button in $block_2$ (defined in ADP_7) is pressed, it generates a message *toggle_attr* which triggers the ADP to hide or show the block $block_1$. This block, however, is redefined in ADP_8: instead of showing three attributes $attr_1, attr_2$, and $attr_3$ it displays $attr_4$.

ADP $ADP_8 : ADP_7$ **Observes** *Composite* : *Type*
    Block $block_1$**:**
        $Composite.attr_4$
End ADP_8

Figure 3.15: An ADP that redefines a block.

3.8 Structural and cross-reference linking

In Hadez there are two types of links: structural and cross-reference. Structural links are determined by the characteristics of the structural graph of the hyperbase of the application. Structural links are used to link the different hyperpages into which a composite is broken.

Cross-reference links, however, are links that do not necessarily follow the structural graph of the hyperbase. A cross-reference link can point from an

arbitrary ADP to another ADP. A major constraint of the structural graph is that it cannot include cycles. Cross-reference linking does not have that restriction.

One of the most important features of Hadez is its independence from the run-time system in which the application is going to be presented. This poses a special challenge when defining links:

- The specification does not know the addressing system of the run-time system. In fact, it does not even know on which run-time system the application will be displayed. This is important, as it does not constrain the specification of the design to characteristics unique to a given run-time system.
- The link destination should, therefore, provide enough information to the run-time system by which it will be possible to identify the destination of the link.

Structural and cross-reference linking in Hadez are implemented by specifying a given ADP and the composite that it observes as the destination of the link. The run-time system is responsible for finding the correct address in which that ADP presents the corresponding composite. As discussed in chapter 2, the same composite can be presented to the reader in more than one hyperpage in the hypermedia application, potentially using the same ADP. In that case, the run-time system non-deterministically chooses one of them .

ADP_{3b}, depicted in figure 3.16, shows the use of a cross-reference link to an ADP_x that observes a composite $Composite_2$. Note how the precondition of the ADP binds the value of one attribute of the object to a specific composite ($Composite_2$).

The main difference between structural and cross-reference linking is how the destination composite is computed. In structural linking, the destination composite is an attribute of the current composite; in this case, the designer has chosen to split the current composite into two or more hyperpages; these links correspond to some arcs in the structural graph of the application. In cross-reference linking the destination composite is not part of the current composite, and it should be computed from the value of the current composite, for example, by finding a composite that satisfies a given relation to the current composite.

ADP ADP_{3b} **Observes** *Composite* : *Type*
Declarations:
$Composite_2 : type_2$
Block $block_1$**:**
anchor $attr_1$ **linkto** $ADP_x(Composite_2)$
Preconditions:
$Composite_2.name = attr_1$
End ADP_{3b}

Figure 3.16: An ADP with non-structural links.

The run-time system is responsible for knowing what it the address of a given hyperpage and how the perspectives of a given application are mapped into these pages. Cross-linking poses the risk of having links to hyperpages that might not be instantiated in the final application. It is the responsibility of the run-time system to remove those anchors that point to locations not available in the current application. This feature has the following advantages:

- The designer does not need to know either the addressing scheme or the final address of the destination of the link.
- The designer does not need to know whether the destination composite—to which the ADP links—is going to be present in the final application. The designer specifies the link and, if it is available, the run-time system activates the anchor, otherwise, it is not presented to the reader. This feature allows flexible instantiation of a subset of the application, in which any links outside this subset are automatically removed, avoiding lack of referential integrity in the final application.

Finally, it is not required that an ADP is already defined before its identifier can be used in a **linkto** statement. It is necessary, however, that the ADP is defined somewhere in the scope of the specification.

3.8.1 Grammar for ADP schemas

Figure 3.17 shows the part of the grammar of Hadez related to the declaration of ADPs.

ADPSchema	::=	**ADP** ADPName (:Ancestor)? **Observes** CompName : CompType *ADPBody* **End** ADPName
ADPBody	::=	*Decl*? *BlockDecl** *Invar*? *Precond*? *MessHandler**
Decl	::=	*type_decl* | *MessageDecl*
MessageDecl	::=	message_name : (**input** | **output** | **internal**) **message**
BlockDecl	::=	**Block:** (*AttrGroup* | *ADPBlock*)
AttrGroup	::=	(attribute_id | *ButtonDecl*)+
ButtonDecl	::=	**button** id → message_id
ADPBlock	::=	**ADP** adp_name
Invar	::=	**Invariants:** *ADPPredicate*+
Precond	::=	**Preconditions:** *ADPPredicate*+
MessHandler	::=	**Message** message_name **:** ADPPredicate
ADPPredicate	::=	(*predicate* | ADPAttach | *ForEach* | *SendMess*)
SendMess	::=	**send** message_name
ForEach	::=	**foreach** variable_id:type **in** ADPPredicate+
ADPAttach	::=	**attach** attribute_id **to** block_id

Figure 3.17: Syntax of ADP Schemas.

3.9 Composition of two or more ADPs

ADP ADP_9 **Observes** *Composite* : *Type*
Declarations:
Visible: boolean
toggle_att: input message
Block $block_1$**:**
$Composite.attr_1$
$Composite.attr_2$
$Composite.attr_3$
Preconditions:
$Visible = TRUE$
Message *toggle_att*:
$Visible \Rightarrow hide(block_1)$
$\neg Visible \Rightarrow show(block_1)$
$Visible' = \neg Visible$
End ADP_9

ADP ADP_{10} **Observes** *Composite* : *Type*
Declarations:
toggle_att: output message
Block $block_1$**:**
$Composite.attr_4$ **button** *toggle_att*
End ADP_{10}

$ADP_{11} \triangleq ADP_9 || ADP_{10}$

Figure 3.18: Using composition to define new ADPs.

Figure 3.18 demonstrates the use of composition. Two ADPs are defined, ADP_9 and ADP_{10}. ADP_9 is capable of handing one input message *toggle_attr*; ADP_{10} generates one output message. The resulting *ADP* has the same funtionality as ADP_6 (defined in figure 3.11 on page 74) and ADP_3 (defined in figure 3.8 on page 70).

Composition allows the designer to reuse ADPs without knowing the internals of the composite. The designer needs only to know its interface, that is, the composite type it observes, and the set of input and output messages that it can receive and generate. The semantics of composition are described in detail in 5.5 (on page 123).

3.10 A more complex example of an ADP

In order to show the power of ADPs to specify sophisticated hypermedia application interfaces we proceed to develop the specification of the virtual tours of the Washington's National Gallery of Art (NGA). Garzotto et al. refered to the Web version of the NGA as "one of the best organized and most enjoyable museum sites" [GMP98].

We will concentrate on the museum's virtual tours. The museum uses a virtual tour to present subsets of its collection to the reader. Figure 3.19 shows a snapshop of one of these tours. In the upper half, the page shows a sample of 6 consecutive works in the tour (a room). The reader can choose to move forward or backwards within the tour (by selecting the "next" and "previous" room button). The bottom half shows a list of all the works in the tour, with hyperlinks for each one of them.

Each tour is composed of a sequence of similar works of art (artifacts). The tour can be modeled with a single ADP *NGATourADP*, that observes a composite *Tour* of type *TourComposite*.

We define two given types: *ICON*, which corresponds to the little icon of an image of a painting, and *STRING*.

$$[ICON, STRING]$$

The only attributes of *Artifact* used in this ADP are its icon and its title.

artifact

$$Icon : ICON$$
$$Title : STRING$$
$$\ldots$$

TourComposite will be composed of a sequence of artifacts and names. Both of these attributes are instantiation parameters of the composite.

$$\Xi TourComposite(Elements : \mathrm{seq}\, artifact, Name : STRING)$$

$$\ldots$$

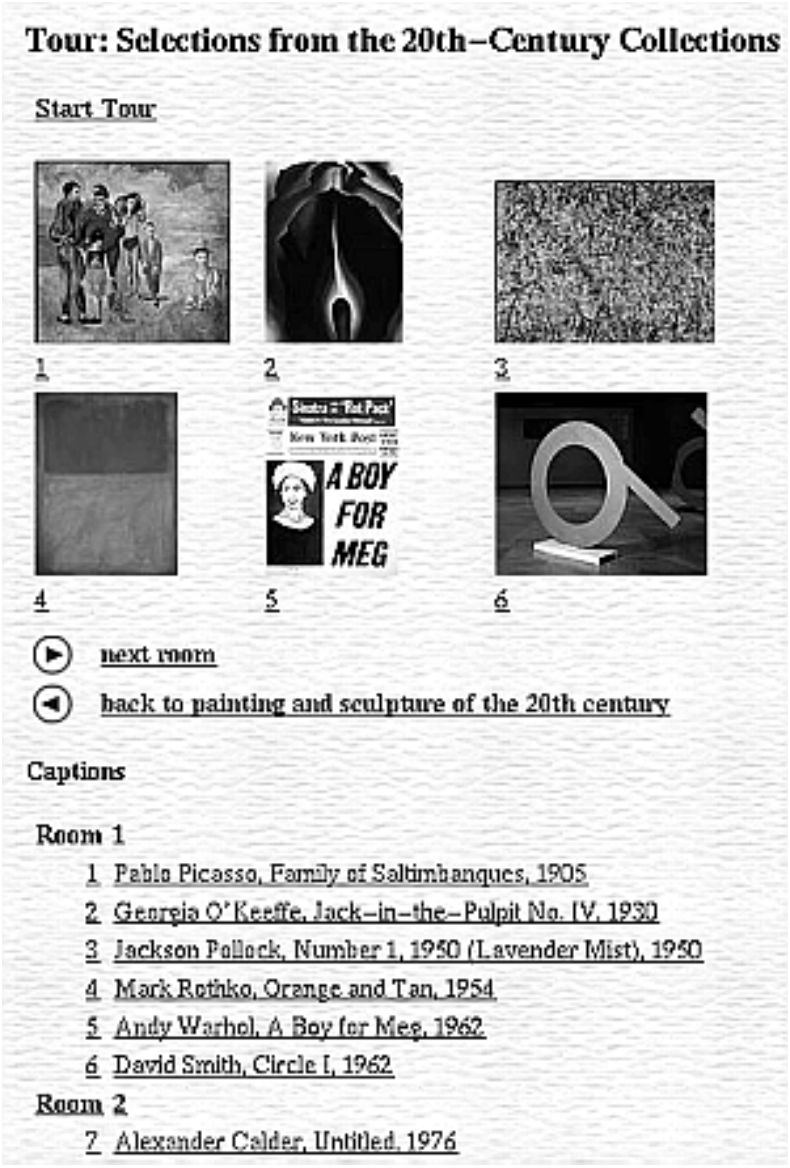


Figure 3.19: NGA web site.

NGATourADP will be divided in four regions, each modeled with a block definition. Figure 3.20 shows the sections of the perspective that will correspond to each block.

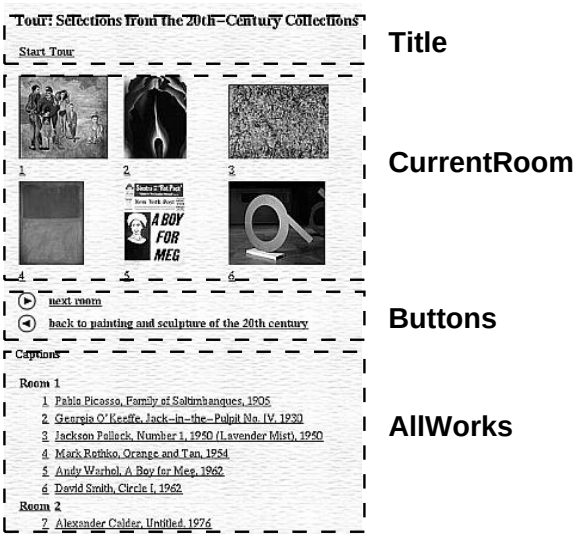


Figure 3.20: Dividing the ADP into regions.

One of the blocks, *CurrentRoom* is going to be modeled with another ADP *ShowRoomADP*.

ShowRoomADP observes a sequence of artifacts. The precondition states that the size of this sequence should not exceed 6 elements. For each one of the elements of the sequence, it creates a hyperlink from its icon to an ADP called *ShowArtifactADP* which is responsible for displaying the respective artifact.

The generic function *subsequence* takes as a parameter a sequence, a starting element, and a number of elements and extracts the corresponding subsequence from its parameter.

$$[X]$$

$$subsequence : \mathrm{seq}_1\, X \times int \times int \rightarrow \mathrm{seq}\, X$$

$$\ldots$$

We now proceed to define the ADP *NGATourADP*. Because “restart”, “next”

ADP ShowRoomADP **Observes** *Works* : seq *artifact*
  Preconditions:
    $\#Works \leq 6$
  Invariants:
    $\forall i : 1..\#Works \bullet$
      anchor $Works(i).icon$ **linkto** $ShowArtifactADP(Works(i))$
End ShowRoomADP

Figure 3.21: The ADP responsible for showing the current room.

and "previous" change only the set of artifacts highlighted in the *CurrentRoom* block, they are specified as buttons that instruct the ADP to change the contents of the CurrentRoom. All three buttons will generate internal messages which are meant to be handled by the ADP.

The first block, *Title* displays only the name of the tour. The second block, *Buttons* is composed of the three buttons of the ADP and attaches the corresponding message to each one of them. The third block, *CurrentRoom*, is described as an aggregated ADP: *ShowRoomADP*. *ShowRoomADP* will be dynamically attached to a subsequence (of length 6) of *Tour.Elements*

The forth and final block, *AllWorks* displays the titles of all the works and links to their corresponding *ShowPaintingADP*.

The preconditions of the ADP state that the tour should have at least one element. It initializes *CurrentOffset* with 1. This variable will be used as the index to the element of the artifacts sequence that should be displayed in the current room. In the preconditions, the *attach* statement dynamically attaches the first 6 elements of the sequence to *ShowRoomADP*.

The three message handlers are very similar. They alter the value of *CurrentOffset* in response to their corresponding message. Note that use *CurrentOffset'*: a variable name followed by ' corresponds to the value of the variable after the sequence of predicates. In this particular case, *CurrentOffset'*, corresponds to the value of *CurrentOffset* after the message handler. For this particular ADP, most of the predicates in each message handler make sure that the "current room" displays a valid subsequence of artifacts.

ADP NGATourADP **Observes** *Tour* : *TourComposite*

Declarations:

CurrentOffset : $\mathbb{N}$
show_next_room : *internalmessage*
show_previous_room : *internalmessage*
restart_tour : *internalmessage*

Block *Title*:

Tour.Name

Block *Buttons*:

button *start_tour* $\rightarrow$ *restart_tour*
button *next_room* $\rightarrow$ *show_next_room*
button *previous_room* $\rightarrow$ *show_prev_room*

Block *CurrentRoom*:

ADP *ShowRoomADP*

Block *AllWorks*:

$\forall i : 1..\#Tour.Elements \bullet$
anchor *Tour.Elements*(i)*.title* **linkto**
ShowPaintingADP(*Tour.Elements*(i))

Preconditions:

$\#Tour.Elements \geq 1$
$CurrentOffset = 1$
attach *subsequence*(*Tour.Elements*, $1, 6$) **to** *CurrentRoom*
hide(*previous_room*)

Message *next_room*:

$\#Tour.Elements \geq CurrentOffset + 6 \Rightarrow$
$CurrentOffset' = CurrentOffset + 6$
show(*previous_room*)
$\#Tour.Elements < CurrentOffset + 12 \Rightarrow$ *hide*(*next_room*)
attach *subsequence*(*Tour.Elements*, $CurrentOffset'$, 6) **to** *CurrentRoom*

Message *prev_room*:

$CurrentOffset > 6 \Rightarrow$
$CurrentOffset' = CurrentOffset - 6$
show(*next_room*)
$CurrentOffset \leq 7 \Rightarrow$ *hide*(*previous_room*)
attach *subsequence*(*Tour.Elements*, *CurrentOffset*, 6) **to** *CurrentRoom*

Message *restart_tour*:

$CurrentOffset = 1$
attach *subsequence*(*Tour.Elements*, $1, 6$) **to** *CurrentRoom*
hide(*previous_room*)

End NGATourADP

Figure 3.22: The Tour ADP.

3.11 Summary

This chapter describes Hadez, the specification language for hypermedia. Hadez is an object-oriented specification language for hypermedia. It is based on Z and is modeled around the Hadez data model described in chapter 2. Hadez has a formal syntax, which guarantees that a specification can be parsed in order to verify whether it is written according to the syntax rules of Hadez. A specification that passes this test is said to be syntactically correct.

A syntactically correct specification should then be verified to be complete. A specification is complete if it is syntactically correct and all the identifiers used in the specification are properly defined. Finally, if a specification is complete, it can by verified to be type consistent. A specification is type consistent if all its constructs do not violate the Hadez typing rules.

Hadez is composed of three main parts: a conceptual schema, a structural schema, and a perspective schema. The conceptual schema is composed of a description of types and instances on those types. The structural schema is a group of composite schemas. A composite schema is a description of how to instantiate higher level composites from instances in the hyperbase (conceptual schema).

The perspective schema is a collection of Abstract Design Perspectives (ADPs). An ADP describes how a reader should perceive a given object and how the interaction with the user alters this perception. An ADP specifies what parts of a composite are shown to the reader, how the composite is broken into hyperpages, and how the interaction with the user changes the reader's perception of the composite. ADPs can be combined into more complex ones by using aggregation, inheritance and composition.

CHAPTER 4

Specifying a Virtual Museum

Museums make very good subjects for the demonstration of design methods and tools. Garzotto and Paolini illustrated HDM using the Microsoft *Art Gallery* [GMP95b]; Rossi and Schwabe demonstrated OOHDM with a Web museum for the works of Candido Portinary [SRB95b]. Museums are composed of well defined entities (artists, artifacts) which are interrelated by simple relationships. Nonetheless, despite their logical simplicity, these entities can be presented in a wide variety of ways, making them a good example. In this chapter, we demonstrate the use of Hadez by writing a specification for the Web museum of the National Gallery of Art.

4.1 The National Gallery of Art web site

The Web museum of the National Gallery of Art Web Museum, in Washington (NGA-WM—www.nga.gov) is one of the best web sites of its type [GMP98]. The NGA-WM main purpose is to display the permanent and temporary collections of the museum. This museum is considered a well-designed hypermedia application.

The main sections of the museum are: *General Information*, *The Collection*, *Exhibitions*, *Online Tours*, and the *Gallery Shop*. In order to keep the size of this specification manageable we are going to specify only the collection and the online tours sections of the museum.

The main entities of the NGA-WM are artifacts, of five different types: paintings, sculptures, decorative arts, works on paper and architecture. The museum also includes information about artists. The core of the museum is a collection of virtual tours. These tours show sets of artifacts. The site includes four different types of tours: *collection tours, in-depth study tours, architecture tours*, and *virtual exhibit tours.* In order to keep the size of this specification manageable, we are not going to include architectural artifacts or tours as part of it. Figure 4.1 is an entity-relationship diagram showing the main classes of the museum and their relationships.

Figure 4.2 shows an OOHDM diagram showing the main navigational contexts created in this application. For instance, from the main page it is possible to access a list of all the schools (a school is a group of artists under a common influence), ordered by type; or a list of all the artists, ordered by name. The artifacts are accessible from either a tour, an artist or a gallery.

4.2 Collections

The museum's collection is composed of art pieces (artifacts). Art pieces can be categorized into four different classifications: paintings, decorative art, sculptures and work on paper. These types are represented by the enumerated type *ARTIFACT_TYPE*.

$$ARTIFACT\_TYPE = (painting, decorativeart, sculpture, workonpaper)$$

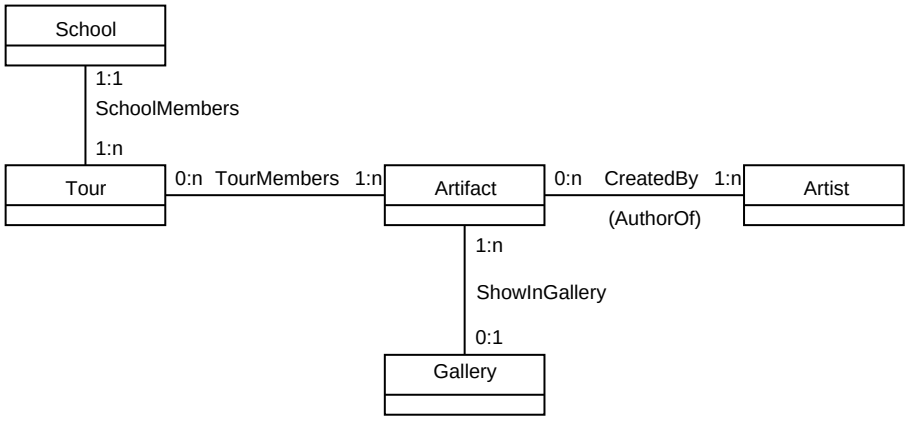


Figure 4.1: Entity-relationship diagram showing the different classes in the museum application and their relationships. The museum is composed of a large group of artifacts, created by artists; the artifacts are shown in tours (virtual tours in the web site) or in galleries (which correspond to the galleries in the real-world museum). The tours are organized in schools (a school is a group of artists under a common influence).

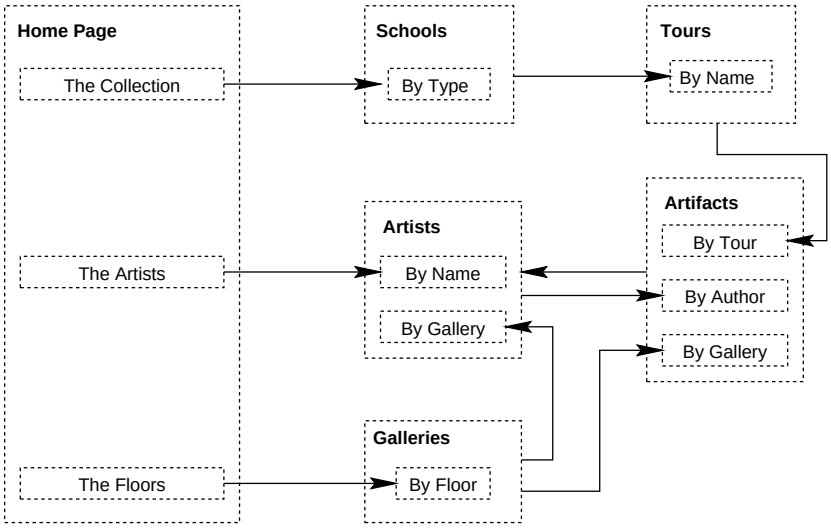


Figure 4.2: OOHDM diagram depicting the navigational properties of the virtual museum.

Several given types will be required by the specification, namely, *TEXT*, used to represent large blobs of text; *DATE* for, as its name implies, date; *IMAGE* which represents a digital photograph; *ACCESSION_TYPE* which represents a unique identifier for an artifact within the museum; *STRING* is used to represent sequences of characters; and *FLOOR_TYPE* which represents the different floors of the two buildings of the museum.

$TEXT, DATE, IMAGE, ACCESSION\_TYPE, STRING, FLOOR\_TYPE$

4.2.1 The classes

The main class of the application is *artifact*. This class is composed of a series of fields that need to be displayed by the application.

Artifact

Name : *STRING*
AccessionNumber : *ACCESSION_TYPE*
TypeOfArtifact : *ARTIFACT_TYPE*
CreationDate : *DATE*
PhysicalDescription : *STRING*
Owner : *STRING*
Photograph : *IMAGE*
Icon : *IMAGE*
Description : *TEXT*
Bibliography : *TEXT*
ExhibitionHistory : *TEXT*
Provenance : *TEXT*
ConservationNotes : *TEXT*
Copyright : *TEXT*
Inscription : *TEXT*

Artists are an important part of the application. Every artifact shows facts about its corresponding artist; furthermore, the museum shows a listing of all artists that have artifacts in the collection. Similar artifacts that do not have a known artist are assigned a "particular anonymous artist". For example, the artist *British 20th Century* gathers all artifacts from anonymous artists in Britain, created during the 20th Century in Britain. These anonymous artists are treated as any other type of artist, in which *BirthYear* and *DeathYear* are empty.

Artist
Name : *STRING*
BirthYear : *STRING*
DeathYear : *STRING*
Nationality : *STRING*
Biography : *TEXT*

Artifacts are always shown as part of a tour. Apparently, the museum only creates tours of artifacts of the same type.

Tour
Name : *STRING*
Overview : *TEXT*
TypeofTour : *ARTIFACT_TYPE*

Tours are collected into groups of similar tours, called "schools". For example, the tours *"Manet and His Influence", "Camille Pissarro, Vincent van Gogh, Paul Cézanne", "Paul Gauguin", "Claude Monet" and "Edgar Degas"* all are part of the school *"French Painting of the 19th Century"*. All tours in a *School* should have the same type.

School
Name : *STRING*
Description : *TEXT*
TypeOfArtifacts : *ARTIFACT_TYPE*

The NGA-WM also displays information about the current, past and future exhibitions on display at the physical museum. These exhibits do not usually have an equivalent Web exhibition; instead, the NGA-WM displays only a textual description of the corresponding exhibitions.

Exhibition
Name : *STRING*
StartingDate : *DATE*
EndingDate : *DATE*
Description : *TEXT*

The museum presents maps of the physical building, allowing the reader to view what is in each of its galleries.

Gallery

$$
\begin{array}{l}
Name : STRING \\
Floor : FLOOR\_TYPE \\
FloorPlan : IMAGE
\end{array}
$$

The museum shows special *virtual exhibitions* on a specialized topic. These exhibitions are different to tours, and they appear to be designed each one at a time. For the sake of space, this specification will not include these exhibitions.

Finally, we define some instances of the previously declared types:

$$
\begin{array}{ll}
Collection : \mathbb{P}\, Artifact & \text{[The entire collection of artifacts]} \\
AllArtists : \mathbb{P}\, Artists & \text{[All the artists displayed in the collection]} \\
AllTours : \mathbb{P}\, Tour & \text{[All the electronic tours of the NGA-WM]} \\
TourOfTheWeek : Tour & \text{[A weekly tour, selected from AllTours]} \\
AllSchools : \mathbb{P}\, School & \text{[All the schools]} \\
AllGalleries : \mathbb{P}\, Gallery & \text{[The set of all galleries]} \\
nAllFloors : \mathbb{P}\, FLOOR\_TYPE & \text{[The set of all floors]}
\end{array}
$$

4.2.2 The relationships

One of the most important relationships is *CreatedBy*, which relates artists with artifacts. Every artifact in the collection has at least one creator, and similarly, each artist might create one or more artifacts. The domain of the relationship is the collection, and its range is the set of artists represented in the museum.

$$
\begin{array}{l}
CreatedBy : Artifact \leftrightarrow Artist \\
\hline
\mathrm{dom}\, CreatedBy = Collection \\
\mathrm{ran}\, CreatedBy = AllArtists
\end{array}
$$

AuthorOf is the inverse of *CreatedBy* and will be useful to find the artifacts created by a particular artist.

$$
\begin{array}{l}
AuthorOf : Artist \leftrightarrow Artist \\
\hline
AuthorOf = CreatedBy\sim
\end{array}
$$

Tours are composed of a non-empty sequence of artifacts. All the artifacts must be of the same type and they must match the type of the tour. The function *TourMembers* maps a given tour to its corresponding sequence of artifacts.

$$
\begin{array}{|l}
TourMembers : Tour \rightarrow \mathrm{seq}_1\, Artifact \\
\hline
\forall\, T : Tour \bullet \\
\quad TourMembers(T) \bullet \\
\quad\quad \forall\, i : 1..\#TourMembers(T) \bullet \\
\quad\quad\quad TourMembers(T)(i).TypeOfArtifact = T.TypeOfArtifact
\end{array}
$$

In a similar way, *SchoolMembers* is a function that maps a given *School* to its component set of *Tour*. Because the component tours do not have to be ordered, the functions return a non-empty set of *Tour*.

$$
\begin{array}{|l}
SchoolMembers : School \rightarrow \mathbb{P}_1\, Tour \\
\hline
\forall\, TG : School;\ TSeq : \mathbb{P}_1\, Tour \bullet \\
\quad SchoolMembers(TG) = Tseq \Rightarrow \\
\quad\quad \forall\, i : 1..\#Tseq \bullet Tseq(i).TypeOfTour = TG.TypeOfArtifacts
\end{array}
$$

Location is a function that maps artifacts to the gallery in which they are located (in the physical galleries of the museum).

$$
\begin{array}{|l}
Location : Artifact \nrightarrow Gallery \\
\hline
\mathrm{dom}\, ShownInGallery \subseteq AllArtifacts \\
\mathrm{ran}\, ShownInGallery = AllGalleries
\end{array}
$$

ShownInGallery is a function that returns the artifacts being displayed in a given gallery. Because these are the physical galleries, its predicate section states that no artifact can be shown in two different galleries.

$$
\begin{array}{|l}
ShownInGallery : Gallery \rightarrowtail \mathbb{P}_1\, Artifact \\
\hline
ShownInGallery = Location\sim
\end{array}
$$

GalleriesInFloor returns the set of galleries that exist on a given floor. It is defined as a injective function and therefore it is not necessary to specify that each gallery can only appear in one floor.

$$GalleriesInFloor : Floor \rightarrowtail \mathbb{P}_1\, Gallery$$

$$\begin{aligned}
&\operatorname{ran} GalleriesInFloor = AllGalleries \\
&\forall F_1, F_2 : Floor \bullet \\
&\quad F_1 \neq F_2 \Rightarrow GalleriesInFloor(F_1) \cap GalleriesInFloor(F_1) = \varnothing
\end{aligned}$$

4.2.3 Creating the composites

We start by defining *ArtistComposite*, which relates artists to their creations:

$$\Xi ArtistComposite(A : Artist)$$

$$Artifacts : \mathbb{P} Artifact$$

$$Artifacts = AuthorOf(\!| \{A\} |\!)$$

One of the most important composites corresponds to an artifact. *Artifact-Composite* gathers a given artifact, and its non-empty set of artists.

$$\Xi ArtifactComposite(Art : Artifact)$$

$$Authors : \mathbb{P}_1\, ArtistComposite$$

$$Authors = \Xi ArtistComposite \circ CreatedBy(\!| \{Art\} |\!)$$

A *TourComposite* gathers all the artifacts that are part of that tour (a non-empty sequence of artifacts).

$$\Xi TourComposite(T : tour)$$

$$Elements : \operatorname{seq}_1 ArtifactComposite$$

$$Elements = \Xi ArtifactComposite \circ TourMembers(T)$$

Similarly, *SchoolComposite* gathers a set of the tours.

$$\Xi SchoolComposite(S : School)$$

$$ToursComp : \mathbb{P}_1\, TourComposite$$

$$ToursComp = \Xi TourComposite \circ SchoolMembers(S)$$

The composite *AllToursComposite* gathers all the *Tours* into a single composite.

$$\Xi AllToursComposite$$
$$Tours : \mathbb{P}\, TourComposite$$
$$Tours = \Xi TourComposite \circ AllTours$$

Similarly, the composite *AllSchoolsComposite* gathers all the *Schools* into a single composite.

$$\Xi AllSchoolsComposite$$
$$Schools : \mathbb{P}\, SchoolComposite$$
$$Schools = \Xi SchoolComposite \circ AllSchools$$

The *IndexOfArtists* is a set of all the artists which have at least one creation in the collection.

$$\Xi IndexOfArtists$$
$$ArtistsComposites : \mathbb{P}\, ArtistComposite$$
$$\exists ArtistsWithWorks : \mathbb{P}_1\, Artist \bullet$$
$$\quad \forall A : Artist \bullet AuthorOf(\!| \{A\} |\!) \neq \varnothing \Rightarrow A \in ArtistsWithWorks$$
$$\quad ArtistsComposites = \Xi ArtistComposite \circ ArtistsWithWorks$$

GalleryComposite corresponds to a gallery that displays a set of artifacts:

$$\Xi GalleryComposite(G : Gallery)$$
$$DisplayedArtifacts : \mathbb{P}\, ArtifactComposite$$
$$DisplayedAuthors : \mathbb{P}\, ArtistComposite$$
$$DisplayedArtifacts = \Xi ArtifactComposite \circ ShownInGallery(G)$$
$$DisplayedAuthors = \{Author : ArtistComposite \mid$$
$$\quad (\exists A : ArtifactComposite \bullet A \in DisplayedArtifacts \land Author \in A.Art.Authors)\}$$

FloorComposite will include all the galleries in a given floor:

$$\Xi FloorComposite(Floor : FLOOR\_TYPE)$$
$$Galleries : \mathbb{P}_1\, GalleryComposite$$
$$Galleries = \Xi GalleryComposite \circ GalleriesInFloor(Floor)$$

And finally, *AllFloorsComposite* gathers all the floors in the museum.

$$
\begin{array}{|l}
\hline
\Xi AllFloorsComposite \\
\hline
Floors : \mathbb{P}_1\ FloorComposite \\
\hline
Floors = \Xi FloorComposite \circ AllFloors \\
\hline
\end{array}
$$

The *CollectionComposite* is composed of the tour of the week plus a set of all the tours in the virtual museum.

$$
\begin{array}{|l}
\hline
\Xi CollectionComposite \\
\hline
WeekTourComposite : TourComposite \\
AllSchools : AllSchoolsComposite \\
\hline
WeekTourComposite = \Xi TourComposite(TourOfTheWeek) \\
\hline
\end{array}
$$

The *MainPageComposite* consists of: an artifact, selected at random from the collection (as a result, the image in the main page of the museum changes randomly); the entire collection of schools; the floors of the building; and index of artists.

$$
\begin{array}{|l}
\hline
\Xi MainPageComposite \\
\hline
RandomArtifact : ArtifactComposite \\
TheCollection : CollectionComposite \\
TheFloors : AllFloorsComposite \\
TheArtists : IndexOfArtists \\
\hline
\exists\, Random \in Collection \bullet \\
\quad RandomArtifact = \Xi ArtifactComposite(Random) \\
\hline
\end{array}
$$

4.3 Describing the perspectives

The presentation of composites is done with abstract design perspectives (ADPs). We first declare two perspectives, which will be defined later. By declaring them here, an ADP can refer to one of these ADPs, allowing circular references. For example, an *ArtifactADP* needs to link to *ArtifactPhotoADP*

$$ArtifactADP : ADP(ArtifactComposite)$$
$$ArtistADP : ADP(ArtistComposite)$$

4.3.1 The artifact's perspectives

Artifacts are shown in two different ADPs: *ArtifactPhotoADP*, which shows a large photo, intended to fill most of the screen, along with the basic information about the artifact; and *ArtifactADP*, a detailed page that displays all the information known about the artifact.

Figure 4.3 shows *ArtifactPhotoADP*. It displays a photograph of the painting followed by some of the attributes of the artifact, including the name of each of its authors.

ADP ArtifactPhotoADP **Observes** *A* : *ArtifactComposite*
  Invariants:
    A.Art.Photograph
  Block *GeneralInfo*:
    foreach *Auth* : *Author* **in** *A.Authors* **do**
      Auth.Name
    A.Art.Name
    A.Art.Owner
    A.Art.CreationDate
    A.Art.AccessionNUmber
    anchor "Information" **linkto** *ArtifactADP*(*A*)
End ArtifactPhotoADP

Figure 4.3: ArtifactPhotoADP is intended to display a large photo and basic information about an artifact.

Figure 4.4 corresponds to the main view of an artifact. In the declarations and preconditions of the ADP, the variable *Tours* is defined. *Tours* will contain the set of tours in which the artifact is displayed. In the block *ToursOfWhichItIsPart*, the ADP will display the name of each one of those tours in which this artifact is presented. The icon in the artifact links to the *ArtifactPhotoADP* which observes the same *ArtifactComposite*. In the block *AuthorsInfo*, the ADP displays information related to each one of the authors of the painting, including a link to their corresponding *ArtistADP*.

The NGA-WM uses buttons to hide and show information from a given page. For instance, the ADP shows only one of the following attributes at a given time: artifacts description, bibliography, exhibition history, conservation notes, and provenance; button—each corresponding to each attribute—selects which one should be the only one displayed.

ADP ArtifactADP **Observes** A : *ArtifactComposite*
Declarations:
$Tours : \mathbb{P}\, Tour$
show_desc : **internal message**
show_bibl : **internal message**
show_prov : **internal message**
show_hist : **internal message**
show_cons : **internal message**
Preconditions:
$Tours = \{t : Tour \mid A.Art \in \text{ran}(TourMembers(t))\}$
Block *GeneralInfo*:
anchor *A.Art.Icon* **linkto** *ArtifactPhotoADP*(*A*) *A.Art.Name*
A.Art.PhysicalDescription
A.Art.Owner
A.Art.CreationDate
anchor "Full screen image" **linkto** *ArtifactPhotoADP*(*A*)
Block *AuthorsInfo*:
foreach *Auth* : *Artist* **in** *A.Authors* **do**
anchor *Auth.Name* **linkto** *ArtistADP*(*Auth*)
Auth.BirthYear
Auth.DeathYear
Block *ToursOfWhichItIsPart*:
foreach *t* : *Tour* **in** *Tours* **do**
t.Name
Block *Description*:
A.Art.Description
Block *Bibliography*:
A.Art.Bibliography
Block *ExhibitionHistory*:
A.Art.ExhibitionHistory
Block *ConservationNotes*:
A.Art.ConservationNotes
Block *Provenance*:
A.Art.Provenance
Block *Buttons*:
button *desc* → *show_desc*
button *bibl* → *show_bibl*
button *prov* → *show_prov*
button *hist* → *show_hist*
button *cons* → *show_cons*

Continued in next page...

Figure 4.4: ADP for Artifacts.

...continued from previous page

Preconditions:
hide(*Bibliography*)
hide(*ExhibitionHistory*)
hide(*ConservationNotes*)
hide(*Provenance*)

Message *show_desc*:
show(*Description*)
hide(*Bibliography*)
hide(*ExhibitionHistory*)
hide(*ConservationNotes*)
hide(*Provenance*)

Message *show_bib*:
hide(*Description*)
show(*Bibliography*)
hide(*ExhibitionHistory*)
hide(*ConservationNotes*)
hide(*Provenance*)

Message *show_hist*:
hide(*Description*)
hide(*Bibliography*)
show(*ExhibitionHistory*)
hide(*ConservationNotes*)
hide(*Provenance*)

Message *show_cons*:
hide(*Description*)
hide(*Bibliography*)
hide(*ExhibitionHistory*)
show(*ConservationNotes*)
hide(*Provenance*)

Message *show_prov*:
hide(*Description*)
hide(*Bibliography*)
hide(*ExhibitionHistory*)
hide(*ConservationNotes*)
show(*Provenance*)

End ArtifactADP

Figure 4.5: ADP for Artifacts, continued.

4.3.2 The tour's perspective

We now turn our attention to the description of tours. As it was previously described, a *Tour* is composed of a non-empty sequence of *Artifact*. A tour is presented to the reader in blocks of six artifacts at a time, called a "room". If the tour has more than six artifacts, then it is divided in more than one room. In a *Tour*, the upper part of the screen will be devoted to a room, and this is going to be modeled with the ADP *ShowRoomADP*, which is defined in figure 4.6. This ADP will show an icon of each one of the artifacts linking to their corresponding ADP.

ADP ShowRoomADP **Observes** $Works : \text{seq}_1\ ArtifactComposite$
Preconditions:
$\#Works \leq 6$
TheWorks:
foreach $A : ArtifactComposite$ **in** $Works$ **do**
anchor $A.icon$ **linkto** $ArtifactADP(A)$
End ShowRoomADP

Figure 4.6: The ADP responsible for showing the current room.

The *TourADP*, depicted in figure 4.7, is responsible for showing a *Tour*. It embeds a *ShowRoomADP* in the block *CurrentRoom* and, by using the reserved word **attach**, dynamically changes the subsequence that the current room displays. The variable *CurrentOffset* determines the index to the first element of the room (a room is always of size 6) and the function *subsequence* (defined below) is used to extract, from a sequence, a certain number of elements starting at a given offset.

$[X]$

$$subsequence : \text{seq}_1\ X \times \mathbb{N} \times \mathbb{N} \rightarrow \text{seq}\ X$$

$$\forall\ Seq : \text{seq}_1\ X;\ i, n : \mathbb{N} \bullet$$
$$subsequence(Seq, i, n) = Seq \upharpoonright \{j : \mathbb{N} \bullet j \geq i \land j \leq i + n - 1\}$$

The buttons *start_tour*, *next_room*, and *previous_room* are used to change the value of *CurrentOffset*. When the reader presses the button *next_room*, *CurrentOffset* is incremented by 6 (the size of a room); a new subsequence of artifacts,

starting in the new value of *CurrentOffset* is attached to the ADP *ShowRoomADP*.

The ADP also shows the entire list of names of artifacts and links to their corresponding ADP.

4.3.3 Perspectives for schools and the collection

In the ADP for a school, there is a random icon from one of the artifacts in one of the tours in the school; this icon links to its corresponding *ArtifactADP*. In the next block, there is a description of the school. Finally, it lists each of the name of the tours in the school with a link to their corresponding *TourADP*. Figure 4.8 shows this ADP.

The list of all schools is classified by the type of artifacts that the school displays. As a consequence, it is necessary to select—within a specific school—those groups of artifacts that are of a particular type. The function *SchoolOfType* takes two parameters: a set of *SchoolComposite* and a variable of *ARTIFACT_TYPE* and returns only those elements of the set of the specified type.

$$
\begin{array}{|l}
SchoolsOfType : \mathbb{P}\, SchoolComposite \times ARTIFACT\_TYPE \\
\qquad\qquad\qquad\qquad \rightarrow \mathbb{P}\, SchoolComposite \\
\hline
\forall\, SCSet : \mathbb{P}\, SchoolComposite;\ Ty : ARTIFACT\_TYPE \bullet \\
\quad SchoolsOfType(SCSet, Ty) = \\
\qquad \{Sch : SchoolComposite \mid Sch \in SCSet \land Sch.S.TypeOfArtifacts = Ty\}
\end{array}
$$

Each type of school is presented by the ADP *ListOfSchoolsADP* which observes a set of *SchoolComposite*, all of them of the same type. It first shows the type of the artifacts and then proceeds to list the name of each school, linking to the corresponding ADP.

ListOfSchoolsADP is meant to be included into the ADP *CollectionADP*. *CollectionADP*, defined in figure 4.3.3, will list all the different types of artifacts and for each, select those schools that include artifacts of that type, and display them using the ADP *ListOfSchoolsADP*. The ADP also shows the name of the "Tour of the Week", which links to its corresponding *TourADP*.

ADP TourADP **Observes** *Tour* : *TourComposite*
Declarations:
CurrentOffset : $\mathbb{N}$
show_next_room : *internalmessage*
show_previous_room : *internalmessage*
restart_tour : *internalmessage*
Block *Title*:
Tour.Name
Block *Buttons*:
button *start_tour* $\rightarrow$ *restart_tour*
button *next_room* $\rightarrow$ *show_next_room*
button *previous_room* $\rightarrow$ *show_prev_room*
Block *CurrentRoom*:
ADP *ShowRoomADP*
Block *AllWorks*:
foreach A : *ArtifactComposite* **in** *Tour.Elements* **do**
anchor A.*Art.Name* **linkto** *ArtifactADP*(A)
Preconditions:
$\#$*Tour.Elements* ≥ 1
CurrentOffset $= 1$
attach *subsequence*(*Tour.Elements*, $1, 6$) **to** *CurrentRoom*
hide(*previous_room*)
Message *next_room*:
$\#$*Tour.Elements* $\geq$ *CurrentOffset* $+ 6 \Rightarrow$
CurrentOffset$'$ $=$ *CurrentOffset* $+ 6$
show(*previous_room*)
$\#$*Tour.Elements* $<$ *CurrentOffset*$' + 6 \Rightarrow$ *hide*(*next_room*)
attach *subsequence*(*Tour.Elements*, *CurrentOffset*$'$, 6) **to** *CurrentRoom*

Message *prev_room*:
CurrentOffset $> 6 \Rightarrow$
CurrentOffset$'$ $=$ *CurrentOffset* $- 6$
show(*next_room*)
CurrentOffset$' \leq 1 \Rightarrow$ *hide*(*previous_room*)
attach *subsequence*(*Tour.Elements*, *CurrentOffset*$'$, 6) **to** *CurrentRoom*
Message *restart_tour*:
CurrentOffset $= 1$
attach *subsequence*(*Tour.Elements*, $1, 6$) **to** *CurrentRoom*
hide(*previous_room*)
End TourADP

Figure 4.7: The ADP for a Tour.

ADP SchoolADP **Observes** *Sch* : *SchoolComposite*
Declarations:
RandomArtifact : *ArtifactComposite*
Preconditions:
$\exists\, T : TourComposite \bullet$
$T \in Sch.ToursComposites$
$RandomArtifact \in \operatorname{ran} T.Elements$
Block *Image*:
anchor *RandomArtifact.Art.Icon* **linkto** *ArtifactADP*(*RandomArtifact*)
Block *Description*:
Sch.S.Description
Block *Tours*:
foreach T : *TourComposite* **in** $T \in Sch.S.ToursComp$ **do**
anchor *T.T.Name* **linkto** *TourADP*(T)
End SchoolADP

Figure 4.8: *SchoolADP* is responsible for displaying a group of tours.

ADP ListOfSchoolsADP **Observes** *Sch* : $\mathbb{P}$ *SchoolComposite*
Block *ArtifactType*:
Sch.S.TypeOfArtifacts
Block *GroupOfTours*:
foreach S : *SchoolComposite* **in** *SchoolsComps* **do**
anchor *Sch.S.Name* **linkto** *SchoolADP*(S)
End ListOfSchoolsADP

Figure 4.9: *ListOfSchoolsADP* lists the name of schools and links to their corresponding ADP.

$$
\begin{array}{|l}
RandomArtifacts : \mathbb{P}\, Tour \pfun \mathbb{P}\, ArtifactComposite \\
\hline
\forall\, TS : \mathbb{P}\, Tour \bullet \\
\quad RandomArtifact(TS) = \\
\qquad \{A : ArtifactComposite \mid (\exists\, T : Tour \bullet T \in TS \land A.Art \in T.Elements\} \\
\quad 4 = \#RandomArtifact(TS)
\end{array}
$$

An interesting feature of *CollectionADP* is that it shows the icons of 4 random artifacts in the collection, linking to their corresponding *ArtifactADP*. These random artifacts are chosen using the function *RandomArtifacts*, which chooses 4 random artifacts from a set of *Tour*.

ADP CollectionADP **Observes** *Coll* : *AllCollectionComposite*
  Block *RandomArtifacts*:
    foreach *A* : *ArtifactComposite* **in** *RandomArtifacts*(*Coll.AllTours*) **do**
      anchor *A.Art.Icon* **linksto** *ArtifactComposite*(*A*)
  Block *TourOfTheWeek*:
    anchor *Coll.WeekTourComposite.T.Name* **linkto**
      TourADP(*Coll.WeekTourComposite*)
  Block *Collection*:
    foreach *T* : *ARTIFACT_TYPE* **do**
      T
      ADP*ListOfSchoolsADP*(*SchoolsOfType*(*Coll.AllSchools*, *T*))
End CollectionADP

Figure 4.10: *CollectionADP* shows all the groups of tours in the collection.

4.3.4 The artists' perspectives

The NGA-WM shows a page for each artist who is an author of an artifact in the museum. This pages displays the information known about the artist, and a list of all the artifacts created by her, grouped by type. In a manner similar to the way we defined the ADP for schools, we proceed to define an ADP, *ArtifactsInOrder*, that will observe a given set of type *ArtifactComposite* and show them in alphabetical order. Its local variable *OrderedArtifacts* corresponds to an injective sequence (a sequence with no repeated elements) created from the parameter *Arts*. The ADP assumes that the type STRING is comparable, that is:

$$\_ < \_ : STRING \times STRING$$

$[a < b$ if a goes before b when alphabetically ordered$]$

ADP ArtifactsInOrderADP **Observes** $Arts : \mathbb{P}\, ArtifactComposite$
  Declarations:
    $OrderedArtifacts$: iseq $ArtifactComposite$
  Preconditions:
    $\mathrm{ran}(OrderedArtifacts) = Arts$
    $\forall i : 1..\#OrderedArtifacts - 1 \bullet$
      $OrderedArtifacts(i).Name < OrderedArtifacts(i+1).Name$
  Block *GroupOfArtifacts*:
    foreach $A : ArtifactComposite$ **in** $OrderedArtifacts$ **do**
      anchor $A.Name$ **linkto** $ArtifactADP(A)$
End ArtifactsInOrderADP

Figure 4.11: ADP that shows the name of artifacts, in alphabetical order.

ArtifactsInOrderADP displays, for each artifact composite in the sequence, its name and it links it to its corresponding *ArtifactADP*.

The ADP *ArtistADP* is responsible for showing an *ArtistComposite*. The function *ArtifactsOfArtistOfType* selects the subset of all the artifacts that match a given *ARTIFACT_TYPE*. The block *ArtifactsOfArtist* embeds one *ArtifactsInOrderADP* for each subset of artifacts of a given type.

$$ArtifactsOfArtistOfType : ArtistComposite \times ARTIFACT\_TYPE \rightarrow \mathbb{P}\, ArtifactComposite$$

$$\forall Author : Artist;\ Ty : ARTIFACT\_TYPE \bullet ArtifactsOfArtistOfType(Author, Ty) = \{A : ArtifactComposite \mid A.Art.TypeOfArtifact = Ty \land Author \in A.Authors\}$$

ADP ArtistADP **Observes** *Artist* : *ArtistComposite*
- **Block** *General_Info*:
  - *Artist*.*A*.*Name*
  - *Artist*.*A*.*Nationality*
  - *Artist*.*A*.*BirthYear*
  - *Artist*.*A*.*DeathYear*
- **Block** *ArtifactsOfArtist*:
  - **foreach** *T* : *ARTIFACT_TYPE* **do**
    - *T*
    - **ADP***ArtifactsInOrderADP*(*ArtifactsOfArtistOfType*(*Artist*.*Artifacts*, *T*))

End ArtistADP

Figure 4.12: ADP for *Artist*.

The NGA-WM groups all authors that start with the same letter in a page. *GroupArtistsADP*, shown in figure 4.13, will be the responsible for showing this information. This ADP shows for each author, her name, which links to her corresponding *ArtistADP*. The artists are ordered alphabetically.

ADP GroupArtistsADP **Observes** *Artists* : $\mathbb{P}$ *ArtistComposite*
- **Declarations:**
  - *ArtistsSeq* : iseq *ArtistComposite*
- **Preconditions:**
  - $\mathrm{ran}(ArtistsSeq) = Artists$
  - $\forall i : 1..\#ArtistsSeq - 1 \bullet ArtistsSeq(i).Name <= ArtistsSeq(i+1).Name$
- **Block** *AllArtists*:
  - **foreach** *A* : *Artist* **in** *ArtistSeq* **do**
    - **anchor** *A*.*Name* **linkto** *ArtistADP*(*A*)

End GroupArtistsADP

Figure 4.13: All the artists, in alphabetical order.

Hadez does not have any primitives to handle characters, and, in fact, it does not even define a character type. As a consequence we define an enumerated type for the letters of the alphabet:

$$CHAR = (a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z)$$

We need now to be able to select all artists whose name starts with the given character. We define *StartWith* as a relation, in which a tuple (s, c) is a member of the relation if the string s starts with the character c.

$$StartWith : STRING \leftrightarrow CHAR$$

[If a tuple $(s, c) \in StartWith$ then the string s starts with the letter c]

Now we can define a function that selects, from a set of authors, those authors which start with a given character.

$$AuthorsStartingWith : \mathbb{P}\, ArtistComposite \times CHAR \nrightarrow \mathbb{P}\, ArtistComposite$$

$$\forall\, SA : \mathbb{P}\, ArtistComposite;\ c : CHAR \bullet AuthorsStartingWith(SA, c) = \{Auth : ArtistComposite \mid Auth \in SA \wedge StartWith(Auth.A.Name, c)\}$$

We have everything necessary to describe the ADP that shows all letters of the alphabet and links them to the corresponding set of authors. This ADP is depicted in figure 4.14. Notice that the block *Main* specifies that each letter of alphabet should be displayed, regardless of whether there are any artists under it.

```
ADP AllArtistsADP Observes AllArtists : IndexOfArtists
        Block Main:
                foreach c : CHAR do
                    c
                     anchor c linkto
                         GroupArtistADP(AuthorsStartingWith(AllArtists.ArtistsComposites, A))
End AllArtistsADP
```

Figure 4.14: A list of all characters in the alphabet, linking to their corresponding *GroupArtistsADP*

.

4.3.5 The galleries of the real museum

Each of the galleries displays a list of works that they display, grouped by author. *ExtArtsInOrderADP* is an ADP that uses the ability for an ADP to inherit properties from the ADP *ArtifactsInOrder*. *ExtArtsInOrderADP* redefines the block GroupOfArtists to show not only the name of each of the artifacts, but also its creation date, physical description and accession number.

ADP ExtArtsInOrderADP : ArtifactsInOrderADP **Observes** $Arts : \mathbb{P}\, ArtifactComposite$
  Block *GroupOfArtifacts*:
    foreach A : *ArtifactComposite* **in** *OrderedArtifacts* **do**
      anchor $A.Art.Name$ **linkto** $ArtifactADP(A)$
      $A.Art.CreationDate$
      $A.Art.PhysicalDescription$
      $A.Art.AccessionNumber$
End ExtArtsInOrderADP

Figure 4.15: The ADP *ExtArtsInOrderADP* extends the ADP *ArtifactsInOrderADP*.

Because the artifacts are grouped by their authors, it is necessary to create a function that will return the artifacts for a given author. The function *ArtifactsFromAuthorInSet* is used to find, in a given set of *ArtifactComposite* which ones were created by a given *Artist*.

$$ArtifactsFromAuthorInSet : ArtistComposite \times \mathbb{P}_1\, ArtifactComposite \rightarrow \mathbb{P}_1\, ArtifactComposite$$

$$\forall Arts : \mathbb{P}\, ArtifactComposite;\ Auth : ArtistComposite \bullet ArtifactsFromAuthorInSet(Auth, Arts) = \{A : ArtifactComposite \mid A \in Arts \land Auth \in A.Author\}$$

The *GalleryADP*, defined in figure 4.16, shows all the artifacts in the gallery, ordered by their author. It uses the ADP *GalleryArtistADP* to show each one of the authors, with her corresponding artifacts. Her corresponding artifacts are computed by using the function *ArtifactsFromAuthorInSet*.

In the NGA-WM, a floor is shown with an imagemap with a floor plan of the corresponding floor. Hadez does not have primitives to support imagemaps[1].

[1] Hadez does not support parameters in messages. If parameters are allowed, the imagemap can

```
ADP GalleryADP Observes  Gal:GalleryComposite
        Block GalleryName:
                Gal.G.FloorPlan
                Gal.G.Name
        Block TheArtists:
                foreach Auth : ArtistComposite in Gal.DisplayedAuthors do
                    Auth.A.Name
                     ADP ExtArtsInOrderADP(ArtifactsFromAuthorInSet(
                            Auth, Gal.DisplayedArtifacts))
End  GalleryADP
```

Figure 4.16: ADP for each one of the galleries.

As a consequence, in this specification, the ADP for a floor—*FloorADP*, depicted in figure 4.17—enumerates the names of the different galleries in the corresponding floor (instead of using the imagemap), linking to their corresponding ADPs.

```
ADP FloorADP Observes  Floor:FloorComposite
        Block Name:
                FloorComposite.Floor
        Block TheGalleries:
                foreach Gal : GalleryComposite in Floor.Galleries do
                    anchor Gal.G.Name linkto GalleryADP(Gal)
End  FloorADP
```

Figure 4.17: *ADPFloor* shows all the galleries in a given floor.

The *FloorsADP*, in figure 4.18 shows the all floors. It embeds one *FloorADP* for each one of the floors in the museum buildings.

4.3.6 The main page

Finally, we put everything together by providing access, from the "home page", to all the main sections of the museum. This ADP shows at the top a random

be implemented as a message handler that receives a message which includes the coordinates of the selection. Parameters, however, increase the complexity of Hadez and make its verification more complex. In chapter 6 we introduce a method to extend Hadez with design patterns; the pattern *HyperMap* can be used to implement this feature in the NGA-WM.

```
ADP FloorsADP Observes AllFloors:AllFloorsComposite
        Block TheGalleries:
                foreach F : FloorComposite in AllFloors.Floors do
                        ADP FloorADP(F)

End FloorsADP
```

Figure 4.18: *ADPFloors* shows all the floors in the museum.

image and links it to its corresponding *ArtistADP*. It also provides links to the entire collection, to the floor maps, and to the index of artists.

```
ADP HomePageADP Observes Home : MainPageComposite
        Block RandomImage:
                anchor Home.RandomArtifact.Icon linkto
                        ArtifactADP(Home.RandomArtifact)
        Block Collection:
                anchor "The Collection" linkto
                        CollectionADP(Home.TheCollection)
        Block Buildings:
                anchor "What is on display at the Museum" linkto FloorsADP
        Block Authors:
                anchor "Index of Authors" linkto AllArtistsADP
End HomePageADP
```

Figure 4.19: The main page of the museum.

Due to space limitations, this specification describes only a subset of the NGA-WM. The sections described herein attempt accurately reflect the current version of the museum (as of Spring 2000). The sections of the Web museum not specified here include the *In Depth Tours* and the *Virtual Exhibitions* which are tours that are designed individually and share few common features (for example, the tours "Thomas Moran, His Life and Works" and "Van Gogh's Van Goghs: Masterpieces from the Van Gogh Museum, Amsterdam"), the calendar of events and the electronic store.

4.4 Extending the specification

In order to illustrate the use of inheritance, we present in this section an extension of the museum specification that exploits the object oriented features of Hadez.

Let us assume that the museum differentiates between different types of objects. For example, paintings and photographs, to name a few. The painting adds two attributes, *Technique* and *Material*, while photographs have three more: *Film, ExposureData* and *PrintingData*. We can declare these classes as subclasses of *Artifact*:

Painting : *Artifact*
Technique : *STRING*
Material : *STRING*

Photograph : *Artifact*
Film : *STRING*
ExpositionData : *TEXT*
PrintingData : *TEXT*

Both *Painting* and *Photograph* can be used anywhere an *Artifact* is used (they are descendents of it). Therefore, a tour can also include photographs, or paintings; the tour, however will perceive them as artifacts and their extra attributes will not be visible to it.

In order to exploit the extra attributes of the new class we need to extend the *ArtifactComposite* and the *Artifact* related ADPs in order to handle the classes.

For example, to present *Photograph* it is necessary to extend *ArtifactComposite* by declaring *PhotographComposite* as its child; in this particular case, the new composite does not require any further attributes, and hence, its body is empty.

Ξ*PhotographComposite*(*Art* : *Photograph*) : *ArtifactComposite*

We can then enhance the ADP *ArtifactPhotoADP* by adding a new block that shows the photo related information.

ADP PhotographPhotoADP:ArtifactPhotoADP **Observes** *A* : *PhotographComposite*
Block *PhotoInfo*:
A.Art.Film
A.Art.ExpositionData
A.Art.PrintingData
End PhotographPhotoADP:ArtifactPhotoADP

Figure 4.20: *PhotographPhotoADP* extends the functionality of *ArtifactPhotoADP*.

4.5 Summary

This specification describes precisely and unambiguously the National Gallery Museum. This specification starts by describing the underlying classes of the museum: artifacts, artists, tours, galleries, etc. and relations that correlate them, such as: *CreatedBy*, that relates artifacts with artists; *TourMembers*, which relates tours with its component artifacts; *SchoolMembers*, which relates schools with its component tours; etc.

These classes and relations are used to create higher level entities such as *ArtifactComposite*, which gathers an artifact and its authors; or *TourComposite*, which combines a tour object with the its component set of *ArtifactComposite*.

Finally, the abstract design perspectives, or ADPs, describe how the composites are to be presented to the reader. They describe which attributes of the composite are presented, and how they are linked to other APDs. For instance, the specification describes how a tour is presented in "rooms", each of 6 artifacts. The user is allowed to move forward and backwards through the rooms in order to view the icons of the artifacts. Each of the icons links to the another ADP that describes in detail the artifact. The user is also presented with a list of all the names of the artifacts in the tour.

This specification can be analyzed and verified, which is the subject of the next chapter.

The truth exists—only fictions are invented.

Georges Braque
Pensées sur l'Art, 1917-1955

CHAPTER 5

Verifying the Specification

5.1 Introduction

Formal specification languages offer several advantages over informal specification languages, such as:

- There is evidence that the use of a specification language with well defined syntax and semantics forces the designer to be more careful in the description of a system, hence increases the quality of the design. As Bertrand Meyer stated "formal notations naturally lead the specifier to raise some

questions that might have remained unasked, and thus unanswered, in an informal approach" [Mey85].

- Some properties of the specification can be verified.

The basic idea behind verification is that the specification can be translated into a set of logic statements that can be used to prove or disprove properties about the specification. We are interested in answering questions about the characteristics of the hypertext application described by the design, and more particularly, characteristics of the potential browsing sessions that readers could encounter.

An Hadez specification describes two main types of facts of an application: its structural characteristics, and its behavior. The structural facts describe how the composites and, in essence, what data is included in the application. The behavior of the application relates to the way that the application reacts to the reader input.

5.2 What properties does the specification fulfill?

One of the goals of a formal specification language is the ability to derive new facts from the ones stated in the specification. These new facts are derived through the application of inference rules. We say that we prove a property of the specification if this property is derivable, by using a well defined set of inference rules, from the original specification, and this property is not explicitly stated in the specification.

There are different types of properties that we might be interested to prove in a Hadez specification:

5.2.1 Is the application realizable?

An application is realizable if there exists at least one potential implementation of it. In particular, in Hadez, there are several ways in which an application might become non-realizable. A contradictory specification is non-realizable. In Hadez, there are multiple places in which contradictions can be added to a specification. For example, a perspective specification includes a set of invariants that should always hold at any state of the application. We are interested

in verifying that, by reacting to user requests, the state of the perspective does not violate its invariants. A contradiction might be present in the invariant section of a perspective; for example, the invariant might specify that an attribute A should be visible and invisible at the same time. In other cases, the contradiction only occurs under certain circumstances; for example, a perspective P observes a artifact composite A and includes an invariant that says that the title of A should always be visible; A transition in P that indicates that the title should be hidden is a clear violation of this invariant. The existence of this transition states that the specification has a contradiction if that transition can take place during a browsing session. In this case we are interested in knowing whether there are browsing sessions that can lead to contradictions. We might also be interested to know if the structural schema is realizable. In this case we would like to verify that every composite schema predicate section is satisfiable, that is, there is a potential set of a values for its variables that makes the predicate true.

5.2.2 Is the specification type consistent?

Like the language Z, a specification in Hadez should be statically type consistent. Type inconsistencies in the specification usually lead to errors.

5.2.3 How does the specification behave?

We are also interested in answering questions about how the application behaves when interacting with a given user. These questions can be grouped in two main categories:

- Perspective behavior. Once a given perspective is presented to the reader, how does it react to the reader's actions? In particular: a) what potential states can be reached in a given perspective; and b) for each of those states, what attributes of its observed composite, what buttons and which anchors are shown to the reader.

- Application behavior. In this case, we are interested in facts that hold for the behavior of the entire application. These facts can be either about its navigation—how the reader can move from one perspective to another; or invariants of the application—facts that are true during the entire life of the application.

In order to illustrate how Hadez is useful for verifying properties of the design, we pose some questions on the specification of the virtual museum (as described in chapter 4). We will further discuss these questions and formally specify them in the next sections.

Perspective Behavior

1. Does the *ArtifactADP* perspective always show the name of the artifact? (defined in figure 4.4 on page 99).
2. Does the *ArtifactADP* perspective show at any time the *Provenance* of the artifact?
3. Can the *TourADP* perspective show all the elements of its observed *TourComposite* to the reader? (defined in figure 4.7 on page 103).

Application Behavior

4. Whenever an image of an artifact is displayed, is its title also displayed?
5. Is every artifact in the museum reachable from the root node of the museum?
6. Every time that an attribute of an artifact photo is shown to the reader, is the perspective corresponding to the artifact immediately reachable by following one hyperlink?

In order to answer these questions it is necessary, first, to create an analyzable model of the specification as described by Hadez; and second, rewrite each question into a predicate that can be proved or disproved against the aforementioned description.

5.3 Modeling the specification

From the point of view of Hadez, a hypermedia application is a collection of perspectives that observe a set of composites. The perspectives interact with the reader and this interaction determines how the composites are presented.

From the point of view of the reader, the hypermedia is a collection of nodes which are connected through hyperlinks. From the point of Hadez, however, each node is a perspective, which in turn can be composed of one or more perspectives. The reader interacts with the perspective, by either changing its state (by pressing buttons) or by following hyperlinks, replacing the currently displayed perspective with a new one.

In order to be able to analyze how perspectives interact with a reader, it is necessary to provide a model that can represent their behavior. The model we will use is I/O automata [LT87, LT88]. See appendix B for an overview of I/O automata.

5.3.1 Modeling a perspective

Informally, an I/O automaton is a possibly infinite state automaton. Its reacts to its environment by accepting input messages. An input message forces the I/O automaton to change its state and generate an output message in response. We are particularly interested in finding an equivalent I/O automaton for each perspective; we refer to it as the characteristic I/O automaton of a perspective. The state of the automaton is defined by the internal value of the attributes of the perspective and by which attributes of its observed composite are visible. The state of the perspective, as well as the state of the automaton, can only change by reacting to messages. The perspective's input and output messages (follow hyperlink, pressed button, display anchor, etc.) correspond to input and output messages of its corresponding I/O automaton. For an I/O automaton, an alternating sequence of messages and states is known as an execution fragment. An execution sequence that starts in a starting state is known as an execution of the automaton.

The problem of analyzing how a perspective behaves under the reader requests becomes a problem of analyzing the potential executions of its characteristic I/O automaton. We can therefore ask questions about the behavior of the perspective and translate them into predicates involving its characteristic I/O automaton. For example, the question number 1 "Does the *ArtifactADP* perspective always show the name of the artifact?" can be translated to "for any state in any execution of the characteristic I/O automaton of an *ArtifactADP* perspective, is the title of the artifact visible?", or its negative counterpart "is it true

that there is no state in the execution of the characteristic I/O automaton of an *ArtifactADP* perspective such that the title of the artifact is not visible?". In other words, the executions of the automaton are equivalent to the behavior of the perspective and it is desirable to be able to find, for any question about the the behavior of a perspective, an equivalent predicate on the execution of its characteristic I/O automaton.

We use HTL* to specify the behavioral properties of a perspective and the complete application. HTL* is a first order temporal logic developed by Stotts et al. [SF98]. HTL* contains path quantifiers to express properties that hold over any potential browsing session. We describe in this chapter how the semantics of an atomic predicate of HTL* are adapted in order to be applied to Hadez specifications.

Question 1, for example, can then be rewritten as $\overrightarrow{\forall} \Box A.Art.Name$, where the variable $A.Art.Name$ corresponds to the name of the artifact in the scope of *ArtifactADP* (as defined in figure 4.4 on page 99). The predicate $\Box A.Art.Name$ means the attribute $Name$: of $A.Art$ is visible at subsequent state of the of the perspective *ArtifactADP*. The operator $\overrightarrow{\forall}$ states that it should hold for any potential browsing path.

5.3.2 Modeling the behavior of an entire application

When different perspectives form nodes of a hypermedia application, we are interested in how the reader interacts with it and moves through it. The reader starts at the root node of the application and, by following hyperlinks, can move from one perspective to another. We are, therefore, interested in answering questions regarding the way an application can be browsed.

In order to make more manageable the problem of analyzing the behavior of an application, we impose one restriction in the run time system: there exists only one active viewport at any given time. In other words, only one perspective is visible at any point in a browsing session. A navigable node, from the point of view of the reader, is a perspective. When a reader selects a hyperlink which destination is not the current perspective, the new perspective replaces the current perspective in the viewport. This restricts the potential states of the application. We will argue further below that this framework can be applied to systems that allow more than one viewport.

In order to model the behavior of the entire application, we create an I/O automaton whose behavior corresponds to the behavior of the hypertext application. It is then possible to evaluate HTL* on the potential execution of the I/O automaton of the hypertext application.

For example, we can specify question 4 as: $\forall A : \mathit{Artifact} \bullet \Box A.\mathit{Photo} \Rightarrow A.\mathit{Title}$; which means that it is always true that for any artifact, if its photo is displayed then its title should also be displayed at the same time.

5.4 Formalizing perspectives

We now proceed to formalize the execution of perspectives in order to be able to specify and prove properties about their execution. We will start by defining some concepts: the set of attributes of a composite and the visibility of a composite.

5.1 Definition (Set of Attributes of a Composite) *The set of attributes of a composite C, denoted by $\mathcal{A}(C)$ is defined, recursively as the singleton $\mathcal{A}(C) = \{C\}$ if C is an atomic composite, and as $\mathcal{A}(C) = \{C\} \cup \bigcup \mathcal{A}(c_i)$ for each composite c_i which is an attribute of C. For a hypermedia application A, $C \in \mathbb{C}(A)$, implies that $\mathcal{A}(C) \subseteq \mathbb{C}(A)$; that is, if a composite is part of an application, then all its attributes are also part of the application.*

The set of attributes of a composite will help us define the concept of visibility of a composite. Informally, a composite is visible if at least one element of its set of attributes is visible, at a given state.

5.2 Definition (Visibility of a Composite) *A Composite C is said to be visible at the state s of the execution of a given perspective, denoted by ν(C,s), if it can be perceived by the reader in the current perspective at the state s or there $\exists a \in \mathcal{A}(C)$ s.t. $\nu(a,s)$.*

As the reader interacts with a perspective, the latter might display a given attribute at one time, but not at another, as it reacts to the external messages, sent by other perspectives or by the reader. We will properly define the notion of state of an perspective further bellow.

In section 2.5.5 (on page 36) we defined a perspective as follows:

5.3 Definition (Perspective) *A perspective P is a tuple $P = \langle S, s_0, s_h, C, M, \Psi, V \rangle$ where S is a set of states; s_0 is the starting state for the perspective s.t. $s_0 \in S$; C is a composite being observed by the region; M is a set of messages that the region either handles or generates; Ψ is a relation of the form $\Psi \subseteq S \times M \times S$; and V is a relation of the form $V \subseteq S \times \mathcal{A}(C)$, in which $(s_i, c_j) \in V$ iff $\nu(s_i, j)$ (the component c_j of C is visible at state s_i). There exists one state $s_h \in S$ s.t. $\neg\nu(s_h, C)$, we refer to this state as the hidden state of the perspective.*

A perspective is restricted to only be able to show attributes of the composite that it observes. The local variables defined in the body of an ADP are used to keep track of the state of the perspective and cannot be presented to the reader. S defines a set of potential states of the perspective. For each of these states, the perspective will make visible a given subset of the attributes of the perspective (denoted by the relation V which specifies which attributes of the composite are shown to the reader at a given state). The number of attributes in a composite is assumed to be finite. The messages the perspective can accept (M) in conjunction with the transitions relation (Ψ) determine how the state of the perspective changes, and therefore how it shows or hides attributes of the given composite. The perspective, when first shown to the reader, starts at state s_0, and the state s_h corresponds to the state in which the perspective is not currently shown to the reader.

M is partitioned into three disjoint sets: input, output and internal. We refer to each one of these sets as $input(M), output(M)$ and $internal(M)$, respectively. Input messages are classified in four main classes:

- Display anchor. The perspective should be shown, in its starting state. This is the result of the reader choosing a hyperlink to the current perspective.
- Follow anchor. The reader has chosen a hyperlink from the current perspective.
- Button message. The reader has pressed a button that triggers its corresponding message.
- Perspective to perspective. The perspective receives a message generated from another perspective.

Output messages can be classified as follows:

- Display destination anchor. A reader has created a "follow anchor" message (an input message) but the desired destination anchor is not in the current perspective.
- Perspective-to-perspective. Like its input counterpart, this message is create by a perspective and its recipient is another perspective.

Internal messages are sent and handled internally by the perspective. The messages main objective is to alter the state of the perspective. In order to guarantee that a perspective can always handle any messages it receives, there should exist a transition $(s_i, m, s_j) \in \Psi$ for each $s_i, s_j \in S$ and $m \in M$.

We now procede to establish the relationship between perspectives and I/O automata. Informally, the perspective state corresponds to the state of the I/O automaton. The perspective's messages correspond to the actions in the I/O automaton. The perspectivpe's messages are already partitioned into three sets: input, output and internal. Each will correspond to one partition of the action signature of the I/O automaton. The set of starting states of A is a singleton with s_0 as its element.

5.4 Definition (Characteristic I/O automaton of a perspective) *Given a perspective $P = \langle S, s_0, C, M, \Psi, V \rangle$, we can construct its characteristic I/O automaton $CIO(P)$ as $CIO(P) = \langle actsig, S', S'_0, \Psi', \phi \rangle$ where:*

- $S' = S$
- *actsig = M, and it is partitioned as follows:*

$$input(CIO(P)) = input(M)$$
$$internal(CIO(P)) = internal(M)$$
$$output(CIO(P)) = output(M)$$

- $S'_0 = \{s_0\}$
- $(s_i, m, s_j) \in \Psi \Leftrightarrow s_i \xrightarrow{m} s_j \in \Psi'$

- $\phi = \varnothing$

We can then define an execution of a perspective as follows:

5.5 Definition (Execution of a perspective) *The set of executions of a perspective is the set of executions of its characteristic I/O automaton.*

And by extension, a behavior is:

5.6 Definition (Behavior of a perspective) *The set of behaviors of a perspective is the set of behaviors of its characteristic I/O automaton.*

The concept of execution is useful for the verification of an perspective. Take, for instance, the region A_2 described by ADP_2 (as defined in figure 3.6 on page 69). A designer might be interested to know whether the attribute $attr_3$ is being displayed by this region at any time. The answer can be found by analyzing the characteristic I/O automaton of the perspective: If there exists an execution of the $CIO(A_2)$ with a state in which $attr_3$ is shown, then the attribute is visible at some point; for ADP_2 in particular, that state is S_0—as $show(attr_3)$ is part of the preconditions of the perspective.

Similarly, it is possible to find contradictions in a specification. For example, let's assume that an invariant is added to ADP_2 that indicates that $attr_3$ should always be hidden; any execution of the $CIO(A_2)$ will show that this invariant is violated, as $attr_3$ appears shown in multiple states in any execution of the $CIO(A_2)$.

5.5 Parallel composition of perspectives

Perspectives can be composed into more complex perspectives, as explained in section 3.9 (on page 81). In parallel composition, two perspectives collaborate with each other in order to model a more complex one.

When composing two or more perspectives in parallel, we use their characteristic I/O automaton to describe the semantics of their composition. I/O automata are particularly useful to describe this type of composition.

In order to compose two perspectives A and B ($A \| B$), they should satisfy some conditions:

- A and B should observe the same composite. $A\|B$ will, therefore, observe that same composite.
- The characteristic I/O automaton of A should be strongly compatible with the characteristic I/O automaton of B. We satisfy this condition by requiring that each message generated by a perspective to be unique across the specification. In order to guarantee that a follow anchor and a display destination anchor messages are unique, they will be composed of the name of its source and destination perspectives.

We proceed to define the composition of two perspectives:

5.7 Definition (Composition of two perspectives) *The composition of two perspectives* $A = \langle S_a, s_a, C, M_a, \Psi_a, V_a \rangle$ *and* $B = \langle S_b, s_b, C, M_b, \Psi_b, V_b \rangle$ *is defined as:*

$$A\|B = \langle S_a \times S_b, \langle s_a, s_b \rangle, C, M_{A\|B}, \Psi_{A\|B}, V_{A\|B} \rangle$$

The messages ($M_{A\|B}$) that the resulting perspective can handle are defined as:

- $input(M_{A\|B}) = (input(M_a) \cup input(M_b)) - (output(M_a) \cup output(M_b))$. In other words, those input messages that match an output message of the other perspective are no longer visible to the exterior.
- $output(M_{A\|B}) = output(M_a) \cup output(M_b)$. All output messages remain as such. For example, a button message m might be matched to an input message of the other perspective, forcing a transition in it, but it will still be propagated as an output message of the composed perspective.
- $internal(M_{A\|B}) = internal(M_a) \cup internal(M_b)$. All internal messages remain internal to their own perspectives, and therefore, to the composed perspective.

The transition function $\Psi_{A\|B}$ is created as follows:

$$(\langle s_j[a], s_j[b] \rangle, m, \langle s_k[a], s_k[b] \rangle) \in \Psi_{A\|B} \Leftrightarrow$$
$$\forall\, i \in a, b \bullet (s_j[i], m, s_k[i]) \in \Psi_i \wedge m \in M_i \Rightarrow s_j[i] = s_k[i]$$

In other words, a transition generated by message m in the result perspective should correspond to a similar transition in both A and B created by the same

message; and if m is not part of the messages handled by any of the composing perspectives, then such composite perspective should remain in the same state.

An attribute is visible in the current state of the result perspective if is visible in that state by either A or B:

$$\forall a \in \mathcal{A}(C), s = \langle s_a, s_b \rangle \in S_a \times S_b \bullet \\ \nu(a,s) \in V_{A\|B} \Leftrightarrow \nu(a,s_a) \in V_a \vee \nu(a,s_b) \in V_b$$

5.8 Lemma (CIO of the composition of perspectives) *The characteristic I/O automaton of $A\|B$ is $CIO(A\|B) = CIO(A)\|CIO(B)$.*

Assume $A = \langle S_a, s_a, C_a, M_a, \Psi_a, V_a \rangle$ and $B = \langle S_b, s_b, C_b, M_b, \Psi_b, V_b \rangle$. By definition 5.7, $A\|B = \langle S_a \times S_b, \langle s_a, s_b \rangle, C, M_{A\|B}, \Psi_{A\|B}, V_{A\|B} \rangle$. If A and B are to be composed, both of their CIO are strongly compatible and both perspectives observe the same composite (C). By definition 5.4:

$$CIO(A) = \langle M_a, S_a, \{s_a\}, \Psi_a, \varnothing \rangle$$
$$CIO(B) = \langle M_b, S_b, \{s_b\}, \Psi_b, \varnothing \rangle$$
$$CIO(A\|B) = \langle M_{A\|B}, S_a \times S_b, \{\langle s_a, s_b \rangle\}, \Psi_{A\|B}, \varnothing \rangle$$

By definition B.3 (see page 156 in appendix B):

$$CIO(A)\|CIO(B) = \langle M_a\|M_b, S_a \times S_b, \{\langle s_a, s_b \rangle\}, \Psi_a\|\Psi_b, \varnothing \rangle$$

It can be verified that by construction $M_{A\|B} = M_a\|M_b$ and $\Psi_{A\|B} = \Psi_A\|\Psi_B$. Hence $CIO(\mathrm{A}\|\mathrm{B}) = CIO(A)\|CIO(B)$ □

Under parallel composition, both perspectives behave like a single one, both sharing the current viewport and observing the same composite.

There is another type of composition of perspectives in Hadez: aggregation. In aggregation, one or more perspectives are embedded inside another; the aggregated perspectives observe the current composite or one of its attributes. One particular feature of aggregation is that a given aggregated perspective can observe different composites at different states of the aggregation perspective (by using **attach**). Let us assume that there is an aggregation perspective P which can have as many as five different composites ($C_1, ..., C_5$) attached at different times during the execution of its aggregation perspective P_0. We would

model P as the parallel composition of P observing C_1, P observing C_2, ..., and P observing C_5 plus a requirement that only one of the five perspectives would be visible at any given time. An "'**attach** C_2" statement, for example, will generate a message that hides the currently displayed perspective and shows the perspective that observes C_2.

5.6 The characteristic I/O automaton of a hypertext application

The characteristic I/O automaton of a perspective models its behavior, and the behavior of the perspectives that compose it. An application, however, is a collection of nodes and each one of these nodes corresponds to one perspective. When the reader follows a hyperlink, the currently visible perspective is replaced by the corresponding perspective.

Similar to the way we model the behavior of each perspective by creating its corresponding I/O automaton, we model the behavior of an application with an I/O automaton, which is created by composing the different I/O automata of the perspectives of the application into the characteristic I/O automaton of the application.

Informally, the characteristic I/O automaton of an application is created by composing the characteristic I/O automaton of the different perspectives that compose each of the nodes of the application. The resulting characteristic I/O automaton is the parallel composition of the characteristic I/O automata of the composing perspectives, with some restrictions that reflect the way perspectives are combined into an application. In first place, a follow link message from any perspective makes the starting perspective go into its hidden state, while the destination perspective goes into its starting state. As a result, every time that a link is followed, the current perspective is replaced by the viewport for the destination perspective, while the other perspectives remain in the same state as before the link was followed. The second restriction is that only one perspective is visible at any given time, that is, all but one perspective in the application is in its hidden state. The third restriction is that a button from one perspective cannot trigger a transition of another perspective; that is, buttons

actions have only "node" scope and cannot be received by another perspective corresponding to another node.

5.9 Definition (Characteristic I/O Automaton of an application) *Given a hypertext application composed of a set of perspectives $\mathbb{A} = \{P_0, P_1, ..., P_n\}$, each corresponding to a node in the application, P_O is the root perspective of the application, the characteristic I/O automaton of $\mathbb{A}$, denoted by $\amalg_{\mathbb{A}} = \langle actsig, S, s_0, \Psi, \phi \rangle$ is defined as:*

$$
\begin{aligned}
actsig =& actsig(\|_{j \in 0..n} P_j) \\
& \cup \{m \mid \forall P_j, P_k \bullet \exists \text{ a ``follow anchor'' message } m \text{ from } P_j \text{ to } P_k\} \\
S =& states(\|_{i \in 0..n} P_i) \\
s_0 =& \langle s_0[0], s_h[1], ..., s_h[n] \rangle \\
\Psi =& steps(\|_{i \in 0..n} A_i) \\
& \cup \{(\vec{s_1}, m, \vec{s_2}) \mid \exists i, j \in 0..n \bullet s_1[i] \neq \text{hidden state}(P_i) \\
& \quad \wedge s_1[j] = \text{hidden state}(P_i) \wedge s_2[i] = \text{hidden state}(P_i) \\
& \quad \wedge s_2[j] = \text{start state}(P_i) \wedge m \text{ is a ``follow anchor message'' from } P_i \text{ to } P_j\} \\
\phi =& \varnothing
\end{aligned}
$$

Each $\vec{s} \in S$ can be represented by a tuple $\vec{s} = \langle s[1], s[2], ..., s[n] \rangle$ where $s[i]$ is the state in which the i-th perspective P_i is in the overall state of the application. s_0 is a state such as the root perspective is in its starting state and all the other perspectives are in the hidden state.

This definition states that the action signature of the characteristic I/O automaton of an application is the action signature of the composition of the characteristic I/O automaton of each of the perspectives plus a set of "follow anchor" messages from each perspective to each perspective in the application. This will allow us to link any perspective to any other perspective. The starting state of the automaton is such that the root perspective is in its starting state, but all the other perspectives start hidden. The transitions in the characteristic I/O automaton are such that by acting on a "follow anchor" message, the current perspective goes into hidden state, and the destination perspective goes into its

starting state.

It is necessary to impose one restriction in the perspectives forming $\mathbb{A}$:

$$\forall P_j, P_k \in \mathbb{A} \bullet signature(P_j) \cap signature(P_k) = \varnothing$$

This restriction makes it impossible for one message in the signature of a given perspective to generate a transition in another perspective unless it is a "follow anchor" message (all which are unique across perspectives, because each message contains the source and destination perspective). Its purpose is to restrict the transitions across perspectives to be only those created by a "follow anchor" message. The simplest way to satisfy this requirement, without restricting which perspectives we can compose, is to rename every message created by a button with a unique identifier across $\mathbb{A}$.

We can extend the concepts of behavior of an execution of an application based upon the behavior of perspectives.

5.10 Definition (Execution of an application) *The set of executions of a perspective is the set of executions of its characteristic I/O automaton.*

And by extension, a behavior is:

5.11 Definition (Behavior of an application) *The set of behaviors of a perspective is the set of behaviors of its characteristic I/O automaton.*

5.7 HTL*

HTL* [SFC98] is a first order temporal logic language based on CTL* . It extends CTL* by providing quantifiers over any potential browsing path. In Hadez, the atomic predicates of HTL* are three, each describing a property at a given state of a perspective.

- A perspective P is visible: P. As a corollary, $\neg P$ corresponds to the fact that the perspective is not being displayed in the current state to the reader in the current viewport.
- A perspective P is in its starting state: P_s

- An attribute i of a composite C observed by the perspective P is visible in the current state: $P.C.i$.

The kernel of HTL* is defined as any first order logic formulae on the predicates just described.

HTL* is defined inductively as:

- Every atomic formula p is a HTL* formula.
- If A is a HTL* formula, so are $\neg A$ (not A), $\bigcirc A$ (A holds at the next state after the reference state), and $\Box A$ (A holds at all states after the reference state).
- If A and B are HTL* formulae, so is $A \Rightarrow B$ (implication).

Other commonly used temporal logic operators can be defined in term of these ones:

- $\Diamond A$ (A holds at some state after the current state):
  $\Diamond A \Leftrightarrow \neg\Box\neg A$
- A **until** B (A will hold at the next state until the state in which B holds):
  A **until** $B \Leftrightarrow B$ **atnext** $(A \Rightarrow B) \wedge \bigcirc\Diamond B$

Finally, HTL* defines two quantifiers over any potential execution path:

- $\overrightarrow{\forall} A$ (A holds for all forward paths)
- $\overrightarrow{\exists} A$ (there exists one forward path in which A holds)

5.7.1 Restating properties in HTL*

We can then use HTL* to state formally the questions that were posed in section 5.3.1.

The first three questions can be interpreted in two different ways: on one hand, as properties of a single perspective, independent of the rest of the application. On the other hand, as properties of the entire application.

In the first case, there is an implicit assumption that the perspective starts in its starting state, and is never hidden. In the second, the predicate holds while the perspective is not hidden. We can state this formally as:

5.12 Definition *A predicate p, that holds in a perspective P of type τ, is equivalent to a statement of the form:*

$$\forall P : \tau \bullet P \Rightarrow p$$

where any attribute of P in p should be decorated as an attribute of P.

We can exemplify this definition by formalizing question 1 "Does the *ArtifactADP* perspective always show the name of the artifact?". When interpreted in the scope of the perspective only, we can state it as:

$$\Box A.Art.Title$$

that is, the title of the observed composite A is always visible. The variable A (of type *ArtifactComposite*) is defined in the scope of the perspective, and therefore, it does not have to be declared in the predicate. In the scope of the application, we can write it as:

$$\forall P : ArtifactADP \bullet \overrightarrow{\forall} P \Rightarrow \Box P.A.Art.Title$$

Question 2 "Does the *ArtifactADP* perspective show at any time the *Provenance* of the artifact?" is very similar to question 1 and can be stated, in the scope of the perspective, as:

$$\overrightarrow{\forall} \Diamond A.Art.Provenance$$

Question 3 "Can the *TourADP* perspective show all the elements of its observed *TourComposite* to the reader?" can be specified as:

$$\forall Artif : ArtifactComposite \bullet \overrightarrow{\forall} Artif \in TourComposite.Elements \Rightarrow \Diamond Artif$$

within the scope of *TourADP*; in other words, any *ArtifactComposite* in the elements of the tour of the composite, is eventually shown in the current perspective.

Question 4 "Whenever an image of an artifact is displayed, is its title also displayed?" can be written as:

$$\forall A : Artifact \bullet \overrightarrow{\forall} \Box A.Photo \Rightarrow A.Title$$

Question 5 "Is every artifact in the museum reachable from the root node of the museum?" corresponds to:

$$\forall A : Artifact \bullet \overrightarrow{\exists} \Diamond A$$

other words, every artifact will eventually be visible.

Question 6 " Every time that an attribute of an artifact photo is shown to the reader, is the perspective corresponding to the artifact immediately reachable by following one hyperlink?" can be specified as

$$\forall A : Artifact \ \exists P : ArtifactPerspective \bullet \overrightarrow{\exists} P.A = A \wedge A.Photo \bigcirc P_s$$

that is, for every artifact A, there exists an *ArtifactPerspective* that observes A such that this perspective is visible in the next state. A perspective can only be in its starting state if either the reader follows a hyperlink in which this perspective is the destination or the perspective is part of the current perspective. In either case, the result is the expected one.

5.8 Semantics of HTL*

The semantics of HTL* of can be explained in terms of a *Kripke* structure **K** with an infinite sequence of mappings:

$$\eta_i : \ \mathcal{V} \rightarrow \{t,f\}$$

Where $\mathcal{V}$ is the set of composite identifiers, perspective identifiers or a perspective starting state identifiers. Each one of these corresponds to one of the three atomic predicates of HTL* (as defined in section 5.7). For a composite C, its Kripke structure at state η_i is defined it terms of its visibility at that state:

$$\mathbf{K}_i(C) \Leftrightarrow \exists P, C, C', C \in \mathcal{A}(C') \bullet P \textbf{ observes } C' \wedge \nu(C,s)$$

Where s is the current state of P at η_i. In other words, a composite is visible at a given state if there exists a perspective that observes that C or a composite C' that includes C.

For a perspective $P = \langle S, s_0, s_h, C, M, \Psi, V \rangle$, which is at state $s \in S$ at state η_i

of the application:

$\mathbf{K}_i(P) \Leftrightarrow s \neq s_h$

The value of the P is true if the perspective is not hidden. And for the starting state of P:

$\mathbf{K}_i(P_s) \Leftrightarrow s = s_0$

P_s is true if P is its starting state.

5.9 Verifying a property

There are two ways in which a property can be verified: manually and automatically—using a theorem prover.

5.9.1 Manual verification

In order to illustrate how properties can be verified, we will proceed to prove the predicates corresponding to questions 1 and 2.

5.13 Lemma $(\Box A.Art.Name = \boldsymbol{t})$ *for the perspective ArtifactADP (defined in figure 4.4 on page 99).p*

A quick look a the specification of *ArtifactADP* shows that *Art.Name* is part of the block *GeneralInfo*. The semantics of Hadez indicate that an attribute which is part of a block is always visible as long as the block is visible. There is no predicate that hides *GeneralInfo* in the message handler section of the predicate. Hence *Art.Name* remains visible as long as the perspective is visible. □

5.14 Lemma $(\Diamond A.Art.Provenance) = \boldsymbol{t}$ *for ArtifactADP*

This proof is a little more complex. The only place in which *Art.Provenance* appears is in the block *Provenance*. This block is hidden at the starting state of the application (by using the predicate *hide*(*Provenance*) in the preconditions. Therefore, the predicate $\Diamond A.Art.Provenance$ is equivalent to the predicate: "is the block *Provenance* eventually shown?". The block *Provenance* is shown inside

the message handler for *show_prov*. Therefore, we need to find a state in which a message *show_prov* is generated. *show_prov* can only be generated by the button *prov* which is part of the block *Buttons*. The block *Buttons* is always visible, hence the button *prov* is always available to the reader. Since we assume that the perspective is treated fairly, at some point the reader presses the button *prov* which triggers the message *show_prov*, which in turns shows the block *Provenance*, which contains the attribute *A.Art.Provenance*. □

As the complexity of the predicate and the perspectives increases, it becomes more and more difficult to provide proofs like these. Furthermore, the potential for error in the proofs also increases.

5.10 Automatic verification

The formal framework presented in this chapter can be used to mechanize the verification of properties. In particular, I/O automata have been modeled in Isabelle/HOL [Pau94, NS95b, NS95a]. Z specifications have been translated into HOL and then verified [BG94].

A Hadez specification can be translated into a HOL equivalent. HTL* predicates would then be translated into HOL predicates that can be checked against the specification. Mechanized verification is beyond the scope of this work.

5.11 Summary

In this chapter we have presented a formal framework in which it is possible to specify and verify properties about a Hadez specification. This framework is composed of two main parts: the modeling of perspectives with I/O automaton. The behavior of a perspective is equivalent to the behavior of its corresponding I/O automaton. The verification of a property of the perspective is equivalent to the verification of an equivalent property of the I/O automaton. In second part of the framework, we define HTL*, a temporal logic that is used to state properties of perspectives. HTL* is used to specify formally some questions about the specification of the NGA-WM specification in chapter 4.

Augmenting a method with patterns eases its use by providing higher level constructs. They improve the expressive power of the method.

Rossi et al. [RSL99]

CHAPTER 6

Design Patterns in Hadez

From the appearance of the first hypertext systems [Eng63, Nel65] authors have design and created hypertext and—more recently—hypermedia applications. Authors have also learned what works better under certain circumstances and have gathered a collection of *tricks of the trade* that help them design more effective hypermedia applications. Authors try different approaches to solve a given problem, evaluate what solutions work best and carry on this experience in subsequent designs.

Many problems are ubiquitous and designers are likely to face them eventually. Common sense dictates that designers that face a newly encountered problem should not invent solutions from scratch; rather, they should take advantage

of the knowledge acquired previously by others. A design pattern "describes a problem which occurs over and over again and then describes a solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice" [AIS77]. A design pattern attempts to collect experience from the expert to pass on to other experts or novices in the field, hence avoiding reinvention by others.

Patterns are not unique to the hypermedia world. They were first used in architecture [Ale79] and recently acquired in software engineering design [GHJV, CS95, VCK96]. The first hypermedia design patterns were presented by Rossi [RSG97]; since then many more have been published. They range from generic "golden rules" [NN98] to specialized patterns for collaborative design [SS99].

Appendix A lists all the design patterns published to date. It is based on the list in [GC00] and further expanded to include newer patterns.

6.1 Organizing and classifying patterns into a pattern system

As we argued in [GC00], there is no coherent system of patterns currently available. Most patterns have been presented independent of each other and they do not create a cohesive group. In order for the patterns to be useful, and for them to be integrated into the design process, it is necessary that these patterns conform a pattern system.

Buschmann et al. [BMR+96] defined a pattern system in the scope of software architectures. Their definition can be adapted to hypermedia design: a pattern system is as a collection of patterns, together with guidelines for their implementation, combination and practical use in hypermedia development. A pattern system should satisfy six conditions:

1. It should comprise a sufficient base of patterns.
2. It should describe all its patterns uniformly.
3. It should expose the various relationships between patterns.
4. It should organize its constituent patterns.

5. It should support the construction of hypermedia systems.

6. It should support its own evolution.

The 51 patterns listed in the appendix arguably form a sufficient base of patterns to create a pattern system. Our intention is to create a system that can be incorporated into Hadez.

The first step towards creating a system is to classify the patterns and then organize them into groups. The patterns in each group should share common characteristics. There exist many proposed classifications of design patterns in the literature. Garrido et al. [GRS97] and Rossi et al. [RSG97] divide them based upon the stage of the design process in which they are most likely used: "navigational design" and "interface design", which correspond to the navigational design and interface design stages of OOHDM. In [GC97] we proposed a classification based upon the objective of the pattern: "structural", "presentational", and "support." Bernstein classifies his patterns [Ber98] as rhetorical although they can be perceived as structural. Nanard and Nanard [NN98] proposed a new type of pattern called "golden rules". Lyardet et al. [LRS98a] proposed to classify them based on the issues they target: "information organization", "interface organization" and "implementation". Lowe [Low99] proposed a new category called "process patterns". Each of these papers classifies only the patterns that it presents. None attempts a wider taxonomy. Paolini and Garzotto [PG99] proposed a global classification based upon whether the patterns specify "user requirements", "generic design ideas", and "well defined design ideas"; unfortunately, they did not classify any design patterns. Nanard and Nanard [NN99] proposed a classification based on three major dimensions: "hypermedia design and development", "hypermedia application", and "hypermedia system"; each of these dimensions is further divided into subdimensions (e.g. hypermedia system is divided into architecture, interaction and production). Although they include a list of design patterns, they did not try to classify them.

In Hadez, we will only be concerned with patterns that specify design problems that are specific enough to be used within Hadez context, but generic enough not to be considered implementation specific. These patterns will form a pattern system within the framework of Hadez and therefore be integrated into the development process. Patterns are organized according to the stage of

the design—as defined by Hadez—to which they pertain:

Component Construction. These patterns characterize, mainly, the way that individual composites are created. Examples of patterns in this group are *Node as a single unit, Component Layout*, and *Composite Consistence*.

Structural Design Patterns. In the scope of Hadez, structural design patterns solve problems involving the way groups of composites are interrelated, and therefore, determine the overall structure of the characteristic graph of the hyperbase. Examples of these patterns are *Cycle, Counterpoint, Mirrorworld, Tangle, Sieve, Montage, Missing Link, Neighborhood*, and *Split/Join*.

Perspectives Construction and Navigation. These patterns assist in the creation of ADPs, and how they should be crosslinked. *Active Reference, Navigational Context, Navigational Feint, Navigational Observer, News, Decorator, Set-based Navigation*, and *Landmark* are good examples of patterns in this group.

Behavioral/User Interaction. Patterns that solve problems related to the way the user and the application interact. In the scope of Hadez, they solve problems related to the way the ADPs handle user input. The patterns in this group include *Behavior Anticipation, Information on Demand, Link Destination Announcement, Process Feed-Back*, and *Collector*.

6.2 Integrating design patterns into Hadez

Rossi et al.[RSG97], Schmidt [Sch95] and Garzotto et al. [GPBV99] have argued why design patterns are valuable design artifacts:

- Patterns enable widespread reuse of architectures.
- Patterns improve communication within and across designers and developers.
- Pattern explicitly capture knowledge that experienced designers already understand implicitly.

- Patterns facilitate training of new designers.
- Patterns increase the quality of design.
- Patterns reduce the cost of design and implementation.

The constructs of Hadez allow the designer to add patterns to a specification, and subsequently specify parts of the application in terms of these patterns. Design patterns become reusable solutions ready to be applied, and extend Hadez specification language.

When faced with a problem already solved by a design pattern, the designer only needs to indicate that this part of the application is going to be solved with that pattern. The designer then proceeds to indicate how the participants of the pattern should be instantiated.

The goal of using patterns is to simplify the specification of the design. They indicate how a part of the application is going to be solved in terms of well-known solution.

Take for instance the pattern *set-based navigation* (described in detail in section 6.4 in page 140). This pattern indicates how to solve the problem of presenting a set of related objects to the reader. This problem is a common one in a hypermedia application; in the scope of Hadez, sets of composites are frequently presented to the reader. The designer is confronted with having to specify how each one of these sets is to be navigated and presented to the reader. By using the pattern, the designer can just indicate in the design, for example: *"for this set of composites, use the* set-based navigation *pattern in which you create the menu of the composites using the perspective P_1 and you order the elements according to the relation R."* The job of P_1 is to show one element of the set and link it to the corresponding perspective that shows it in detail.

This description is simple and straightforward and only requires the reader to know what the *set-based navigation* pattern is, and what its participants are.

Hadez attempts to take advantage of this succinctness in the specification by providing a framework in which Hadez can be enhanced with a system of patterns, in which each pattern is clearly and uniformly described, and, more important, characterized in a way that it can be easily used within the specification.

Hadez introduces the notion of design pattern characterization. A design pattern characterization is a description of a pattern and an abstract type signature of its participants.

6.3 Characterizing design patterns

One of Hadez goals is to support the type verification of the specification. Design patterns, however, are not types or classes. Patterns are generic guidelines on how to solve a specific problem. We reach a compromise by only formalizing the parameters necessary to instantiate the pattern.

The instantiation of a design pattern implies knowing what objects will be used as the participants of the patterns and what generic characteristics they should have. A pattern should be described in terms of the type signatures of its participants and the type signature of the result of the instantiation of the pattern. With this information, it will be possible to verify whether the instantiation of a pattern is consistent with its definition.

For example, the main participant of the pattern *set-based navigation* is a group of navigable objects. It will be clearly useless to instantiate this pattern with, say, just one object, or with a set of integers (as the integers, by themselves, are not navigable).

In Hadez, patterns are described with a group of generic types and the type signatures of its participants (in terms of these generic types). The use of generic types in patterns is similar to the way generic composite schemas use generic types (see section 3.4.3 in page 60). As a result, if a pattern characterization indicates that a pattern requires a set of objects, the pattern cannot be instantiated with only one object, for example.

6.1 Definition (Design Pattern Characterization) *A design pattern characterization (DPC) is a tuple* $P = \langle GenTypes, P_1 : t_1, ..., P_n : t_n \rangle$ *where GenTypes is a set of generic types and* $P_1, ..., P_n$*, each of type* $t_i \in GenTypes$*, are the participants of the design pattern P.*

Some patterns might have no participants. In that case, the DPC of the pattern is an empty sequence. A DSP does not indicate how the pattern is to be implemented. This is an important feature, because Hadez should not

constrain the implementation of the application. By using DPCs we can avoid the specification of repetitive low level details without jeopardizing the quality of the design.

A DPC is described in Hadez with a design pattern characterization schema (DPCS). The general form of a DPCS is:

$\pi\ Name[t_1, ..., t_i] : instance\_type$
$P_1 : t'_1$
$P_2 : t'_2$
...
$P_n : t'_n$

Where *Name* is the name of the pattern; $t_1, ..., t_i$ are to generic types used within the scope of the DPCS; *instance_type* corresponds to the type signature of the element created when the pattern is instantiated (it can be a *composite* or a *perspective* with its corresponding parameter sequence, and determines whether the instantiation of the pattern creates a composite or a perspective); $P_1, P_2, ..., P_n$, each of type t'_1, t'_2, t'_n correspond to the participants of the pattern. Each of t'_i can be defined in terms of the generic types $t_1, ..., t_i$.

A DPCS is used in a pattern instantiation by declaring an element as the result of the application of the pattern. For instance:

$$InstantiatedName \equiv Name[t_1, ..., t_i](P_1, P_2, ..., P_n)$$

Where *InstantiatedName* is the name of resulting element, $t_1, ..., t_i$ correspond to the instantiation of the generic types, and $P_1, ..., P_n$ the instantiation participants. This construct is equivalent to create a composite or perpective schema in which the inner details of the schema are hidden, but well known.

6.4 An example

We will illustrate DPCSs with a pattern that are is particularly useful in most designs: *set-based navigation*.

Name: **Set-based navigation**

Intent: The intent of the pattern is to present in order, and in a uniform way, a set of similar objects.

Motivation: Collections of similar objects, presented as a group, appear in almost any hypermedia application. A reader usually needs, either, sequential access to the each of the members of the set, or direct access to one or more of its members.

Applicability: This pattern is applicable in a variety of domains. For example, the set of articles written by a professor, the set of students in a department, the set of class course offerings by a department for a given term, the set of books on sale today, the set of results of a search engine, etc.

Solution: A good solution involves, first, ordering the elements of the set according to a specific property of the elements. Each of the elements will be presented in one hypermedia node in the same uniform manner, reinforcing in the reader the fact that each element is part of the same set. Each of the nodes will be linked to the previous and next one, so the entire sequence can be traversed sequentially. Furthermore, a main table of contents will assist into navigating directly into one particular node. The table of contents will be ordered according to the same ordering function as the order of the sequence.

Participants:

- Set of composites. These are the objects to navigate.
- Ordering relation. A relation used to order the set into a sequence.
- A perspective to use to present each of the elements of the set in the table of contents perspective. Each of these perspectives shows one or more attributes of an element and links it to another perspective that depicts in detail the element.

Collaborations: The ordering relation transforms the set into a sequence. The table of contents is created with a sequence of perspectives, one for each element in the sequence.

Known uses: This pattern is ubiquitous; almost any hypermedia application lists sets of similar objects.

Related patterns: *Guided Tour* (also known as Pure Guided Tour) [GD99], *Hybrid Collection* and *Collection Center* [GPBV99] are based on the notion of *set-based navigation* and can be considered refinements of it. *Collection Center* [GPBV99] corresponds to the *Menu* component of this pattern.

The DPC of *Set-Based Navigation* is:

$$\pi\ \mathit{Set\text{-}Based\ Navigation}[\mathit{element\_type}] : \mathit{perspective}(\mathbb{P}\,\mathit{element\_type})$$
$$\mathit{OrderRel} : \mathit{element\_type} \times \mathit{element\_type}$$
$$\mathit{ElePers} : \mathit{perspective}(\mathit{element\_type})$$

The first participant of the pattern (the set of navigable objects) is part of the type signature of the instantiated pattern ($\mathbb{P}\,\mathit{element\_type}$), while the other two are part of its participants section.

We exemplify the use of this pattern by defining *GroupArtistsADP* (shown in figure 4.13 in page 107). This perspective shows, for each author, its name; the name is liked to the corresponding *ArtistADP* for that artist.

We need to define a simple ADP that is used to present each artist in the set.

ADP SimpleArtistsADP **Observes** *Artist* : *ArtistComposite*
  Block *ArtistLink*:
    anchor *Artist*.*A*.*Name* **linkto** *ArtistADP*(*A*)
End SimpleArtistsADP

OrderArtistsComposites is a relation that orders elements of type *ArtistsComposite*:

$$\mathit{OrderArtistComposite} : \mathit{ArtistsComposite} \times \mathit{ArtistsComposite}$$
$$\forall a, b : \mathit{ArtistsComposite} \bullet \mathit{OrderArtistComposite}(a, b) \Rightarrow a.A.\mathit{Name} < b.A.\mathit{Name}$$

We are now ready to instantiate the pattern:

$$\mathit{GroupArtistsADP}_2 \equiv \mathit{Set\text{-}Based\ Navigation}[\mathit{ArtifactComposite}]$$
$$(\mathit{OrderArtistComposite}, \mathit{SimpleArtistsADP})$$

Which creates a perspective $\mathit{GroupArtistsADP}_2$ which is equivalent to *GroupArtistADP*.

6.5 A system of patterns for Hadez

The creation of a complete system of patterns in beyond the scope of this thesis. It is our intention, however, to incorporate some design patterns into Hadez. It is also clear that not all patterns can be characterized within this framework because they might be too abstract or their application might not result in a perspective or a composite.

The following patterns are good candidates to be part of our system of patterns:

- Active Reference. It provides a perceivable and permanent reference about the current status of navigation [RSL99]. [GRS97, LRS98b, RSG97].
- Navigational Context. It provides the user with closed navigational subspaces containing related information [GRS97, LRS98b, RSG97].
- Landmark. It provides a set of hyperlinks to important areas of the application [RSL99].
- News. It provides a list of most recently modified items in the application [LRS98a, RSL99].
- Shopping Basket. Keeps track of user selections during navigation, making these selections persistent to process them when the user decides to. Decouple product selection from product consumption and/or processing [RSL99].

This list does not pretend to be definite. As more patterns are discovered and many are further refined, this list will change. The Gamma et al. collection (or *Gang-of-Four patterns*, the first widely accepted catalog of design patterns for software engineering) went through a period of evolution from their first appearance, in 1991, to their final publication, in 1995. For instance, some patterns were added; some had their names changed; and the average size of each pattern grew from two to ten pages.

Furthermore, Hadez gives designers the ability to define and add their own, application domain patterns.

6.6 Summary

In this chapter we integrate design patterns into Hadez. A pattern is characterized by describing the type signature of each of its participants, and the type signature of the result of the instantiation of the pattern (either a composite or a perspective). We use the pattern Set-Based Navigation as an example; we first characterize this pattern, and then instantiate it into a perspective equivalent to one depicted in the specification of the virtual museum.

None but those who have experienced them can conceive of the enticements of science. In other studies you go as far as others have gone before you, and there is nothing more to know; but in a scientific pursuit there is continual food for discovery and wonder.

–Dr. Frankenstein
Mary Wollstonecraft Shelley
Frankenstein, 1818

CHAPTER 7

Conclusions

7.1 Summary

In this dissertation we described Hadez, a formal specification language for hypermedia design. A Hadez specification is divided in three parts: conceptual schema, structural schema, and perspective schema.

- The conceptual schema of Hadez is a collection of given types, type constructors, classes definitions, relations and instances. This part of Hadez is very similar to a Z structural specification (as opposed to a behavioral

one). Hadez allows the declaration of classes that can be defined using single inheritance.

- The structural schema is a collection of Hadez composite schemas. A composite schema describes how to create a composite from other composites or from data from the conceptual schema. As described in section 2.4 (in page 30), a composite schema is composed of three main parts: its parameters section, which specifies what composites or data are required to instantiate the composite; a sequence of free variables; and a set of predicates that bind those free variables to instances in the hyperbase or to other composites.

- The perspective schema is composed of a sequence of abstract perspective schemas (ADP). An ADP serves three main purposes: 1) it specifies how the composite should be broken into pages; 2) it indicates which attributes of the composite should be presented to the reader; and 3) it specifies what attributes of the composite should be hyperlinked to other composites. A perspective is not an static entity. The user can change its state. An ADP specifies how the perspective reacts to the user requests. The state of an ADP determines what parts of the composite the ADP should show at that particular state.

We demonstrated how Hadez can be used in the specification of a large hypermedia system by describing the web site of the National Gallery of Art, in the United States (www.nga.gov). This web site is one of the best of its kind.

We also presented a framework in which questions about the specification can be formalized and verified, in particular, those regarding the behavior of ADPs. Finally, we presented a method to enhance Hadez with the notion of hypermedia design patterns.

7.2 Contributions

The main contribution of this dissertation is Hadez, a formal language for the specification of large hypermedia applications.

- The data model of Hadez divides a hypermedia application in three main components: data (conceptual schema), structure (structural schema) and navigation/presentation (perspective schema). As a consequence, any changes in one of these three components will have a reduced impact on any of the other components.
- A Hadez specification describes unambiguously the characteristics of an application in an implementation-independent manner.
- One of the most important contributions of Hadez is the notion of Abstract Design Perspective (ADP). An ADP is a user interface to a composite and indicates: 1) how a composite is broken into pages; 2) what attributes of the composite are presented to the reader; 3) any crosslinking to other perspectives; and 4) how the interaction with the user affects the behavior of the perspective.
- We have presented a framework for the specification of properties of the specification and their verification. This provides assurance to the designer that the specification complies with certain user requirements. In particular, we argue how the behavior of perspectives can be modeled and analyzed.
- The museum specification (in chapter 4) showed that Hadez can be used to specify large, real-world applications.
- Finally, we demonstrated how the language can be extended with the use of design patterns, which simplifies specifications and promotes the reuse of designs.

Hadez is particularly useful in the specification of data-intensive applications. In the case of hypermedia literature, where entities are difficult to identify ore are non-inexistent, its value will be very limited.

Many hypermedia designers have no logic training. For them, writing a Hadez specification would be very difficult. However, by forcing a designer to write a Hadez specification of the application before it is implemented, will most likely guarantee that the designer has a clear understanding of the requirements of the problem and its proposed solution.

7.3 Future work

The work presented in this dissertation can be extended in several directions, which can be classified in the following categories:

- Tool support for the development of Hadez specification.
- Studies on the effectiveness of Hadez.
- Tool support for mechanical verification.
- Tool support for the automatic implementation of Hadez specifications.
- A further extension of Hadez to support applications that modify the underlying hyperbase

Tool support for Hadez can take several forms. Syntax and type verification require a parser that takes as input a Hadez specification and verifies if it is syntactically correct and type-consistent. The parser could be based in the `FuZZ` Z type checker (recently open sourced) [Spi92b]; or in the PRECC compiler-compiler for which there exists a Z grammar [BB95]. Another area of importance for tool support is the creation of a graphical environment-oriented towards the creation of Hadez specifications. This environment would include syntax and type verification. Finally, a tool for the visualization of the specification would create a graphical representation of the specification (for example, its structural graph) that could be browsed by the designer.

Studies in the usability and adequacy of Hadez for the specification of large actual applications are required. These studies could reveal advantages and disadvantages of the language and would influence its future development and application. It is also necessary to asses the impact of using Hadez in the design process (as compared to current practices). For instance, it would be valuable to know whether the use of Hadez in a large project reduces the overall development time, reduces the number of errors in the implementation, simplifies communication amongst parties involved, eases maintenance and improves the reuse of the design in new applications. Similar studies are needed to asses the impact of using Hadez within the framework of a design method, such as OOHDM or RMM.

The mechanization of the verification of Hadez specifications is an important area for further research. It is important for the user to be able to pose questions to a prover and know the answer with little or no effort. The theorem prover HOL seems like a good candidate for such a system because, both, Z [BG94] and I/O automata [Pau94, NS95b, NS95a] have been modeled with it.

In terms of implementation, it is important to investigate the automated transformation of the specification into an implementation. This transformation would require that primitives in Hadez are translated to primitives in the corresponding run-time system where the application is expected to be displayed. Because a Hadez specification is implementation-dependent, this transformation might not result in a complete implementation. It is also possible to translate the specification into a metadescription which is more implementation-oriented, such as XML. For example, the WebComposition Markup Language (WCML) [GWG97] is an XML-based language oriented towards the implementation of hypermedia applications, particularly in the scope of the World-Wide Web. WCML follows the design process of OOHDM and, as a consequence, it is viable to translate Hadez specifications into WCML.

Finally, Hadez can be extended to allow the modification of the hyperbase. In our current model an application is a view that does not alter the underlying data. Although most hypermedia applications fall into this category, the ones that alter the data are becoming more frequent. Hypermedia is becoming a common interface to database applications in which the readers are allowed to change information. In the future, the difference between a software system and a hypermedia system will be so minute that—for practical purposes—the two systems may be considered to be the same.

APPENDIX A

Summary of Z Syntax

This section summarizes the Z notation used in this paper. The formal semantics of the Z specification language is based on first-order predicate logic and set theory. A complete description of the language can be found in Spivey [Spi92a].

Declarations

The basic unit of modularity is the *schema*. A schema, which may be parameterized by a set of types (i.e., a generic schema), introduces a set of variables and

the predicates constraining their possible values:

SchemaName
variables
---
predicates

GenSchemaName$[FORMALPARM_{1..n}]$
variables
---
predicates

Schema variables are not visible outside the definition, however, a definition can incorporate the declarations and constraints of other schemas using *schema inclusion*:

SchemaName
S
other_variables
---
predicates

where S is a schema defined previously. The declarations of both schemas are merged, and their predicates are conjoined. Schemas may also be defined as the result of schema expressions:

$$SchemaName \mathrel{\widehat=} S \land T \qquad\qquad SchemaName \mathrel{\widehat=} S \lor T$$

These definitions result in a new schema whose signature consists of the signatures of both S and T, and whose predicates are joined by the respective logical connective.

Global variables and axioms are introduced with an *axiomatic* description:

global variables
---
predicates

A *generic constant definition* is similar to a schema, but without a name. The constants are introduced into the global specification scope:

$[FORMALPARM_{1..n}]$
constants
---
predicates

$[X, Y]$	Introduces *given* sets, i.e., X and Y are uninterpreted types.
$X == Y$	The identifier X is a syntactic abbreviation for the expression, Y.

For the following definitions, assume that P and Q are predicates, S, T, and U are sets, X and Y are arbitrary types, and R and Z are relations:

Logic

$P \land Q$	Conjunction
$P \lor Q$	Disjunction
$P \Rightarrow Q$	Implication
$P \Leftrightarrow Q$	Equivalence (if and only if)
$\exists x : S \bullet P$	There exists at least one element of S that satisfies P
$\forall x : S \bullet P$	All elements of S satisfy P

Set Theory

$\{x_1, x_2, ..., x_n\}$	Set enumeration
$\mathbb{P} S$	The power set (set of all subsets) of S
$\mathbb{F} S$	The set of all finite subsets of S
$x \in S$	Membership
$x \notin S$	Non-membership
$\#S$	Cardinality of S
$x \subseteq S$	Subset
$\varnothing$	Empty set
$\{x : S \mid P\}$	The elements of S that satisfy P
$S \cap T$	The intersection of S and T
$S \cup T$	The union of S and T

Relations

$X \times Y$	Cartesian product
$X \leftrightarrow Y$	Binary relation
(a, b)	Ordered pair
first	First of an ordered pair
second	Second of an ordered pair
$\mathrm{dom}\, R$	Domain of R
$\mathrm{ran}\, R$	Range of R
$R \fatsemi Z$	The composition of R and Z, e.g., $\{a : A, b : B, c : C \mid (a, b) \in R \land (b, c) \in Z \bullet (a, c)\}$
$R(\!\mid S \mid\!)$	Relational image of R, e.g., $\{a : A, b : B \mid (a, b) \in R \land a \in S \bullet b\}$

Functions

$X \pfun Y$	The set of all partial functions from X to Y
$X \fun Y$	The set of all total functions from X to Y
$X \psurj Y$	The set of all partial surjections from X to Y
$X \surj Y$	The set of all total surjections from X to Y
$X \pinj Y$	The set of all partial injections from X to Y
$f(a)$	Function application
$f \oplus \{(a, b)\}$	Function override, e.g., $f(a) = b$ replacing any previous value of $f(a)$

Sequences

$\mathrm{seq}\, X$	Finite sequence of X
$\langle\,\rangle$	Empty sequence
$\langle S, T\rangle$ partitions U	$U = S \cup T \land S \cap T = \varnothing$

Important Identities

$$X \rel Y \quad == \quad \mathbb{P}(X \times Y)$$
$$x \mapsto y \quad == \quad (x, y)$$

APPENDIX **B**

Overview of I/O Automata

An I/O automaton [LT87, LT88] is a possible infinite state automaton where each transition between states is labelled from a set of actions. For an I/O automaton A, we refer to its set of actions as $actions(A)$. This set is partitioned into sets of input, output and internal actions. We refer to these sets as $input(A)$, $output(A)$, and $internal(A)$. These sets of actions determine an interface between the automaton and its environment. This interface is known as the action signature of an automaton, $actsig(A)$. An I/O automaton A is formally defined as:

B.1 Definition (I/O automaton) *An I/O automaton A is defined as a tuple $A = \langle actsig, S', S'_0, \Psi', \phi \rangle$ where actsig is an action signature, S' a set of states, $S'_0 \in S'$ is a set of start states, Ψ' is a transition relation $\Psi' \subseteq S' \times actsig \times S'$. ϕ is an equivalence relation used to define fairness.*

An element $(s,a,t) \in \Psi'$ is known as a step and it is represented by $s \xrightarrow{a} t$. An execution fragment consists of a finite or infinite sequence $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} ... \xrightarrow{a_{n-1}} s_n...$ such that $s_i \xrightarrow{a_i} s_{i+1} \in \Psi'$. An execution is an execution fragment that begins with a start state. The execution set of A is denoted by $execs(A)$. A state is reachable if it is the final state of a finite execution.

By removing the actions from an execution fragment of an I/O automaton A we obtain a *schedule* of A. If, from a schedule of A, all the internal actions are removed, the resulting sequence is known as a *behavior*. The set of all behaviors of an I/O automaton represents the externally observable activities that A can execute and it is denoted as $behaviour(A)$.

B.1 Composition

For composition, an output action a of one automaton is matched to automata with input action a. In order for two automata to be composed, their signatures should be *strongly compatible*. A countable collection of action signatures $\{S_i\}_{i \in I}$ is said to be *strongly compatible*, if, $\forall i,j \in I, i = j$:

1. $output(S_i) \cap output(S_j) = \varnothing$
2. $internal(S_i) \cap actsig(S_j) = \varnothing$
3. No action is contained in infinitely many sets $actsig(S_i)$

These restrictions state that a set of I/O automata are strongly compatible if no two automata generate the same output message; the internal actions of one automaton does not trigger an action in another automaton; and, no action is in an infinite number of action signatures of the component I/O automaton.

In order to describe the composition of a set of I/O automata, we need first to describe the action signature of the composition.

B.2 Definition (Action Signature of the Composition of I/O automata) *The action signature S of the I/O automaton $A = \|_{i \in I} A_i$ of a countable collection of strongly compatible I/O automaton $\{A_i\}_{i \in I}$, each with action signature S_i is:*

$$input(S) = \bigcup_{i \in I} input(S_i) - \bigcup_{i \in I} output(S_i)$$

$$output(S) = \bigcup_{i \in I} output(S_i)$$
$$internal(S) = \bigcup_{i \in I} internal(S_i)$$

Notice that the composition does not hide output messages that match input messages of another automaton in the composition. This is because if these messages were hidden, then the order of the composition of more than two automata would generate different resulting I/O automaton. For instance, assume that and I/O automaton A generates an output message M that is an input message of two I/O automata B and C. If matching messages were hidden, then A composed with B would hide B as in internal message and M would never visible if the resulting I/O automaton is later composed with C; on the other hand, if we compose B and C first and then the result I/O automaton with A, both B and C would be able to see the message M generated by A. This is clearly not the desired behavior. It is important that the composition of several I/O automaton be independent of the order in which the composition is made.

Finally, the composition of I/O automata is defined as:

B.3 Definition (Composition of a set of I/O automata) *The composition A of a countable collection of n strongly compatible I/O automaton $A = \|_{i \in 1..n} A_i$ is defined as:*

$$\begin{aligned}
signature(A) =& \|_{i \in 1..n} signature(A_i) \\
states(A) =& \prod_{i \in 1..n} states(A_i) \\
start(A) =& \prod_{i \in 1..n} start(A_i) \\
steps(A) =& \{(\vec{s_1}, a, \vec{s_2}) \mid \forall\, i \in 1..n,\, a \in actions(A_i) \Rightarrow (\vec{s_1}[i], a, \vec{s_2}[i]) \in steps(A_i) \wedge \\
& \quad a \notin actions(A_i) \Rightarrow s_1[i] = s_2[i]\}
\end{aligned}$$

In the above definition, $\prod$ refers to the set-theoretic product of a family of sets. The j-th state of the result automata $\vec{s_j}$ corresponds to the tuple $\langle s_j[1], ..., s_j[n] \rangle$ in which $s_j[i], i \in 1..n$ correspond to the j-th state of the i-th I/O automaton of the composition. The composition $\|_{i \in I} A_i$ can be written as $A_1 \| ... \| A_n$.

APPENDIX C

Published Hypermedia Design Patterns

The list has been ordered by pattern name and includes a brief description and the publications in which it has appeared. In the case of those patterns which have been published more than once by the same group of authors under the same pattern name (for example, *Active Reference* and *News*), we assume that the new versions are refinements of the same pattern, hence we consider it to be the same one. The descriptions presented are the same as in the original papers (in some cases, they were abbreviated for the sake of space). As a consequence, some descriptions are vague, others are stated as a question, and overall, there is no consistency from one pattern description to another.

Active Reference: Provides a perceivable and permanent reference about the current status of navigation. [GRS97, LRS98b, RSG97]

Avatar: How can a self-representation of users be provided in an intuitive way? [SS99]

Behavioral Anticipation: How do you indicate the effect or consequence of activating an interface object? [GRS97]

Behavioral Grouping: How to organize the different types of controls in the interface so the user can easily understand them? [GRS97]

Clustering: Avoid the presentation of more than 7 items simultaneously. [NN98]

Collector: How to make a set of elements behave in the same way depending on one element. [DM99]

Communication Channel: How can information be exchanged that is not directly related to the document content? [SS99]

Component Layout: Several artifacts need to be arranged in respect to their audio-visual properties. [CL98]

Compound: How to describe the resulting behavior when two elements are joined to work together. [DM99]

Constructive Templates: It is a generic specification which makes it easier for the developer to build up actual hypermedia structure and populate it with its data. [NN98]

Contour: Cycles overlap on each other, allowing free movement from one cycle to another. [Ber98]

Counterpoint: Two "voices" alternate, interleaving, giving the reader the option to either follow one or to jump from one to the other. [Ber98]

Cycle: The reader returns to a previously visited node and departs along a new path. [Ber98]

Decorator: Provides a flexible alternative to subclassing for extended functionality. [GLWG99]

Dynamic Configuration Pattern: How to provide the user with the means to perform a selection over a set of options that might be arbitrarily large, while keeping track of them, and then validate them. [LRS98a]

Glue: Joins a number of multimedia artifacts into a single composite artifact. [CL98]

Group Location Awareness: How can we provide a permanent reference about the user's current locations in the collaborative hypermedia space? [SS99]

Hierarchical Structure through Navigation Side Bars: Provides a way to graphically distinguish between hierarchical structure and cross-references when there is only one underlying link type available, as on the Web. [Oes99]

Hyper-Book: Presents a hypertext version of a sequential document (book, article, report). [GC99]

Hyper-Map: Provides an interface to geographical information. [GC99]

Information Factoring: Presents information needed by the reader to understand a given topic/information unit. [LRS98b]

Information on Demand: Lets users decide which items they want further described in the context of the same node. [GRS97, LRS98b, RSG97]

Information-Interaction Coupling: How do we make clear what is the object affected by a control in a node's interface? [GRS97]

Information-Interaction Decoupling: How do you differentiate contents and various types of controls in the interface? [GRS97]

Landmark: Provide direct access to critical sub-systems in the WIS. [RSL99]

Link Creation Method: When is it better to create static links, and when is it preferable to create links through computations? [GRS97]

Link Destination Announcement: Avoids unnecessary link firing by providing information about the destination. [NN98]

Logical Glue: Small information sets need to express meaningful structure to avoid being perceived as an arbitrary grouping. [NN98]

Logical Glue Consistency: Homologous strategies should be used in similar parts of the design in order to help the reader build up a mental model of the structure. [NN98]

Mirrorworld: Provides two or more views of the same information. [Ber98]

Missing Link: Suggests a link that does not exist. [Ber98]

Montage: Several distinct writing spaces appear simultaneously, maintaining their separate identities. [Ber98]

Navigational Context: Provides the user with closed navigational subspaces containing context-related guidelines and relationships. [GRS97, LRS98b, RSG97]

Navigational Feint: Establishes the existence of a navigational opportunity that is not meant to be followed immediately. [Ber98]

Navigational Observer: Decouples the navigation process from the perceivable record of the process. [RSG97]

Neighborhood: Establishes an association among nodes through proximity, shared ornament, or common navigational landmarks. [Ber98]

News: Allows easy access to new information items as the WIS grows. [LRS98a, RSL99]

Node Creation Method: When is it better to create nodes statically, and when is it preferable to create nodes dynamically? [GRS97]

Node as a Single Unit: How do you decide the extent of a node? [GRS97, LRS98b]

Partitioned Incremental Development: Provides the basis for development of hypermedia in an incremental manner, supporting progressive integration and delivery of components. [Low99]

Process Feed-Back: How do we keep users informed about the status of the interaction in such a way that they know what to expect? [GRS97]

Session: How can we structure collaboration between users and groups of users? [SS99]

Set-based navigation: Organizes the information in sets of related information items. Provide intra-set navigation capabilities [RSL99]

Shopping Basket: Keeps track of user selections during navigation, making these selections persistent to process them when the user decides to. Decouple product selection from product consumption and/or processing. [RSL99]

Sieve: Sorts readers through one or more layers of choice in order to direct them to a given section. [Ber98]

Split/Join: Knits two or more sequences together. [Ber98]

Tangle: Confronts the reader with a variety of links without providing clues to guide the reader's choice. [Ber98]

Template: A need exists to produce a collection of composite artifacts similar in structure and contents. [CL98]

User Role: How to represent the different behaviors a user shows, depending on the collaborative context? [SS99]

Virtual Product: Displays a product as part of an electronic catalog. [GC99]

Virtual Room: How can we structure collaboration between users and groups of users in a natural and intuitive way? [SS99]

Bibliography

[AC96] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, New York, 1996.

[AIS77] C. Alexander, S. Ishikawa, and M. Silverstein. *A Pattern Language*. Oxford University Press, 1977.

[Aks96] M. Aksit. Separation and composition of concerns in the object-oriented model. *ACM Computing Surveys*, 28(4es):148, December 1996.

[Ale79] C. Alexander. *A Timeless Way of Building*. Oxford University Press, 1979.

[AMM97] P. Atzeni, G. Mecca, and P. Merialdo. Design and maintenance

of data intensive Web sites. Technical Report RT-DIA-25-1997, Universita della Basilicata, June 1997.

[BB95] P. T. Breuer and J. P. Bowen. A concrete grammar for Z. Technical Report PRG-TR-22-95, Oxford University Computing Laboratory, UK, September 1995. Presented as a poster at the FME'96 symposium [GW96].

[Ber98] M. Bernstein. Patterns of hypertext. In *Proceedings of the Ninth ACM Conference on Hypertext*, Hypermedia Application Design, pages 21–29, 1998.

[BG94] J. P. Bowen and M. J. C. Gordon. Z and HOL. In J. P. Bowen and J. A. Hall, editors, *Z User Workshop, Cambridge 1994*, Workshops in Computing, pages 141–167. Springer-Verlag, 1994.

[BI95] M. Bieber and T. Isakowitz. Designing Hypermedia Applications. *Communications of the ACM*, 38(8):26–27, August 1995.

[BMR+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stad. *Pattern-Oriented Software Architecture – A System of Patterns*. John Wiley, 1996.

[BN92] S. M. Brien and J. E. Nicholls. Z base standard. Technical Monograph PRG-107, Oxford University Computing Laboratory, 11 Keble Road, Oxford, UK, November 1992. Accepted for standardization under ISO/IEC JTC1/SC22.

[Bor98] J.L. Borges. *Obras Completas I*, chapter El Jardín de los Senderos que se Bifurcan, pages 472–480. Emecé Editores, 1998.

[Bus45] V. Bush. As we may think. *Atlantic Monthly*, 176(1):101–108, July 1945.

[BV97] M. Bieber and F. Vitali. Toward Support for Hypermedia on the World Wide Web. *IEEE Computer*, 20(1):62–70, January 1997.

[Cat96] R.G.G. Cattell, editor. *The Object Database Standard: ODMG-93*. Morgan Kaufmann Publishers, Inc., 1996.

[CC94] L.M.F. Carneiro-Coffin. *JASMINUM: Joining ADVs and State Machines in a Notation for User-Interface Modelling*. PhD thesis, University of Waterloo, Waterloo, Ontario, Canada, August 1994.

[CdOdCC94] J.-P. Courtiat, R. Cruz de Oliveira, and L. Rust da Costa Carmo. Towards a new multimedia synchronization mechanism and its formal definition. In *Proceedings of the Second ACM International Conference on Multimedia (MULTIMEDIA '94)*, pages 133–140, New York, October 1994. ACM Press.

[CDOS96] J. P. Courtiat, M. Diaz, R.C. De Oliveira, and P. Senac. Formal Methods for the description of timed behaviors of multimedia and hypermedia distributed systems. *Computer Communications*, 19:1134–1150, 1996.

[CES86] E. M. Clarke, E. Allen Emerson, and A. P. Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications: A practical approach. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[CILS93] D.D. Cowan, R. Ierusalimschy, C.J.P. Lucena, and T.M. Stepien. Abstract data views. *Structured Programming*, 14(1):1–13, 1993.

[CL95] D.D. Cowan and C.J.P. Lucena. Abstract Data Views: An Interface Specification Concept to Enhance Design for Reuse. *IEEE Transactions on Software Engineering*, 21(3):229–243, March 1995.

[CL98] J. L. Cybulski and T. Linden. Composing Multimedia Artefacts for Reuse. In *Proceedings of The 4th Pattern Languages of Programming Conference*, 1998.

[CO96] J. P. Courtiat and R. C. De Oliveira. Proving temporal consistency in a new multimedia synchronization model. In *Proceedings of the Fourth ACM Multimedia Conference (MULTIMEDIA'96)*, pages 141–152, New York, NY, USA, November 1996. ACM Press.

[CS95] J. O. Coplien and D. C. Schmidt, editors. *Patterns Languages of Program Design*. Addison-Wesley, 1995.

[CTL+91] M. A. Casanova, L. Tucherman, M. J. D. Lima, J. L. R. Netto, N. Rodriguez, and L. F. G. Soares. The Nested Context Model for Hyperdocuments. In *Proceedings of the Third ACM Conference on Hypertext*, pages 193–200, 1991.

[DAP97] P. Díaz, I. Aedo, and F. Panetsos. Labyrinth, an Abstract Model for Hypermedia Applications. Description of its Static Components. *Information Systems*, 22(8):447–464, 1997.

[dH97] M. d'Inverno and M. J. Hu. A Z specification of the soft-link hypertext model. In J. P. Bowen, M. G. Hinchey, and D. Till, editors, *ZUM'97: The Z Formal Specification Notation, 10th International Conference of Z Users, Reading, UK, 3–4 April 1997*, volume 1212 of *Lecture Notes in Computer Science*, pages 297–316. Springer-Verlag, 1997.

[Dis99] A. Discenza. Design Patterns for WWW Museum Hypermedia. Technical Report 99.4, Politecnico di Milano, 1999.

[DM99] A. Diaz and R. Melster. Patterns for Modelling Behavior in Virtual Environment Applications. In *2nd Workshop in Hypermedia Development: Design Patterns in Hypermedia*, 1999.

[DP94] J. Dospisil and T. Polgar. Conceptual Modelling in the Hypermedia Development Process. In Jeanne W. Ross, editor, *Proceedings of the 1994 ACM SIGCPR Conference*, pages 97–104, New York, NY, USA, March 1994. ACM Press.

[dP95] M. d'Inverno and M. Priestley. Structuring specification in Z to build a unifying framework for hypertext systems. In J. P. Bowen and M. G. Hinchey, editors, *ZUM'95: The Z Formal Specification Notation*, volume 967 of *Lecture Notes in Computer Science*, pages 83–102. Springer-Verlag, 1995.

[Eng63] D. C. Engelbart. A conceptual framework for the augmentation of man's intellect. In P. D. Howerton and D. C. Weeks, editors, *Vistas in Information Handling, Volume 1*, pages 1–29. Spartan Books, Washington, D.C., 1963.

[Gar88] P. K. Garg. Abstraction mechanisms in hypertext. *Communications of the ACM*, 31(7):862–870, July 1988.

[GC97] D.M. German and D.D. Cowan. Hypermedia Design Patterns. In *7th. Mini Euro Conference on Decision Support Systems, Groupware, Multimedia and Electronic Commerce*, April 1997.

[GC99] D. M. German and D.D. Cowan. Three Hypermedia Design Patterns. In *2nd Workshop in Hypermedia Development: Design Patterns in Hypermedia*, Febrary 1999.

[GC00] D.M. German and D.D. Cowan. Towards a unified catalog of hypermedia design patterns. In *Proceedings of the 33th Hawaii International Conference on System Sciences*, Jan. 2000.

[GD99] F. Garzotto and A. Discenza. Design Patterns for Museum Web Sites. In *Proceedings of MW'99 – 3rd International Conference on Museums and the Web*, pages 144–153, 1999.

[GGRS00] M. Gaedke, H.-W. Gellersen, G. Rossi, and D. Schwabe. Web Engineering. In *Proceedings of the 33th Hawaii International Conference on System Sciences*, Jan. 2000.

[GHJV] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns: Abstraction and reuse in object-oriented designs. In O. Nierstrasz, editor, *Proceedings of ECOOP'93*, Berlin. Springer-Verlag.

[GLR95] A. Ginige, D. B. Lowe, and J. Robertson. Hypermedia authoring. *IEEE Multimedia*, 2(4), 1995.

[GLWG99] M. Gaedke, F. Lyardet, and H. Werner-Gellersen. Hypermedia Patterns and Components for Building better Web Information Systems. In *2nd Workshop in Hypermedia Development: Design Patterns in Hypermedia*, 1999.

[GMP95a] F. Garzotto, L. Mainetti, and P. Paolini. *Designing User Interfaces for Hypermedia*, chapter Hypermedia Application Design: a structured design. Springer Verlag, 1995.

[GMP95b] F. Garzotto, L. Mainetti, and P. Paolini. Hypermedia Design, Analysis, and Evaluation Issues. *Communications of the ACM*, 38(8):74–86, August 1995.

[GMP98] F. Garzotto, M. Matera, and P. Paolini. To Use or Not to Use? Evaluating Usability of Museum Web Sites. In *MW'98 Museums and the Web*, 1998.

[GPBV99] F. Garzotto, P. Paolini, D. Bolchini, and S. Valenti. "Modeling by Patterns" of Web Applications. In *Advances in Conceptual Modeling: Proceedings of ER'99 Workshops*, volume 1727 of *Lecture Notes in Computer Science*, pages 293–306. Springer-Verlag, 1999.

[GPS91] F. Garzotto, P. Paolini, and D. Schwabe. HDM – A model for the design of hypertext applications. In *Proc. of ACM Hypertext'91*, Hypertext – Integrative Issues, page 313, 1991.

[GPS93] F. Garzotto, P. Paolini, and D. Schwabe. HDM – A model-based approach to hypertext application design. *ACM Transactions on Information Systems*, 11(1):1–26, 1993.

[GRS97] A. Garrido, G. Rossi, and D. Schwabe. Pattern Systems for Hypermedia. In *Proceedings of The 3th Pattern Languages of Programming Conference*. University of Washington, 1997.

[GW96] M.-C. Gaudel and J. Woodcock, editors. *FME'96: Industrial Benefit and Advances in Formal Methods*, volume 1051 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[GWG97] H.-W. Gellersen, R. Wicke, and M. Gaedke. WebComposition: An object-oriented support system for the Web engineering lifecycle. *Computer Networks and ISDN Systems*, 29(8–13):1429–1437, September 1997.

[Hal90] J. Anthony Hall. Seven myths of formal methods. *IEEE Software*, 7(5):11–19, September 1990.

[Hol91] I. M. Holland. *The Design and Representation of Object-Oriented Components*. Ph.D. thesis, Northeastern University, 1991.

[Ier91] R. Ierusalimschy. A method for object oriented specification with VDM. Technical Report 2/91, Puc Rio, Brazil, 1991.

[ISB95] T. Isakowitz, E. A. Stohr, and P. Balasubramanian. RMM: A methodology for structured hypermedia design. *Communications of the ACM*, 38(8):34–44, August 1995.

[Kro87] F. Kroeger. *Temporal Logic of Programs*, volume 8 of *EATCS Monographs on Theoretical Computer Science*. Springer Verlag, 1987.

[Lan92] K. C. Lano. Z++. In Susan Stepney, Rosalind Barden, and David Cooper, editors, *Object Orientation in Z*, Workshops in Computing, pages 106–112. Springer-Verlag, Cambridge CB2 1LQ, UK, 1992.

[Lan94] D. Lange. An Object-Oriented Design Method for Hypermedia Information Systems. In *Proceedings of the 28th Hawaii International Conference on System Sciences*, jan 1994.

[Low99] D. B. Lowe. Hypermedia Process Assessment Tasks: Patterns of Inspection. In *2nd Workshop in Hypermedia Development: Design Patterns in Hypermedia*, 1999.

[LRS98a] F. Lyardet, G. Rossi, and D. Schwabe. Patterns for Dynamic Websites. In *Proceedings of The 4th Pattern Languages of Programming Conference*, 1998.

[LRS98b] F. Lyardet, G. Rossi, and D. Schwabe. Using Design Patterns in Educational Multimedia Applications. In *Proceedings of EDMedia'98*, 1998.

[LT87] N. A. Lynch and M. R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In Fred B. Schneider, editor, *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing*, pages 137–151, Vancouver, BC, Canada, August 1987. ACM Press.

[LT88] N. A. Lynch and M. R. Tuttle. An introduction to Input/Output Automata. Technical Memo MIT/LCS/TM-373, Massachusetts

Institute of Technology, Laboratory for Computer Science, November 1988.

[MAM+98a] G. Mecca, P. Atzeni, A. Masci, P. Merialdo, and G. Sindoni. The ARANEUS Web-base management system. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 27(2):544–??, ???? 1998.

[MAM+98b] G. Mecca, P. Atzeni, P. Merialdo, A Masci, and G. Sindoni. From Databases to Web-Bases: The Araneus Experience. Technical Report RT-DIA-34-1998, Universita della Basilicata, May 1998.

[MD99] S. Murugesan and Y. Deshpande. ICSE'99 workshop on web engineering. In *Proceedings of the 1999 International Conference on Software Engineering*, pages 693–694. IEEE Computer Society Press / ACM Press, 1999. Workshop summary.

[Mey85] B. Meyer. On formal specifications. *IEEE Software*, pages 7–26, January 1985.

[Nel65] T. H. Nelson. The hypertext. In *Proceedings International Documentation Federation Annual Conference*, 1965.

[NN98] M. Nanard and J. Nanard. Pushing Reuse in Hypermedia Design: Golden Rules, Design Patterns and Constructive Templates. In *Proceedings of the Ninth ACM Conference on Hypertext and Hypermedia*, pages 11–20. ACM Press, June 1998.

[NN99] J. Nanard and M. Nanard. Toward an Hypermedia Design Patterns Space. In *2nd Workshop in Hypermedia Development: Design Patterns in Hypermedia*, 1999.

[NS95a] T. Nipkow and K. Slind. I/O Automata in Isabelle/HOL. In P. Dybjer, editor, *Proc. of Types for Proofs and Programs, LNCS 996*, 1995.

[NS95b] T. Nipkow and K. Slind. I/O automata in Isabelle/HOL. *Lecture Notes in Computer Science*, 996:101–119, 1995.

[Oes99] K. Oesterbye. Hierarchical structure through navigation side bars. In *2nd Workshop in Hypermedia Development: Design Patterns in Hypermedia*, 1999.

[Pau94] L. C. Paulson. Isabelle: A generic theorem prover. *Lecture Notes in Computer Science*, 828:xvii + 321, 1994.

[PG99] P. Paolini and F. Garzotto. Design Patters for the WWW hypermedia: problems and proposals. In *2nd Workshop in Hypermedia Development: Design Patterns in Hypermedia*, 1999.

[Pra97] B. Prabhakaran. *Multimedia Database Management Systems*. Kluwer Academic Publishers, 1997.

[PTdOM98] F. B. Paulo, M. Augusto S. Turine, M. C. F. de Oliveira, and P. C. Masiero. XHMBS: A formal model to support hypermedia specification. In *Proceedings of the Ninth ACM Conference on Hypertext*, Structural Models, pages 161–170, 1998.

[Ros96] G. Rossi. *An Object Oriented Method for the Development of Hypermedia Applications*. PhD thesis, Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, 1996.

[RSG97] G. Rossi, D. Schwabe, and A. Garrido. Design Reuse in Hypermedia Applications Development. In *Proceedings of the Eighth ACM Conference on Hypertext*, Hypertext Design, pages 57–66, 1997.

[RSL99] G. Rossi, D. Schwabe, and F. Lyardet. Improving Web information Systems with Navigational Patterns. In *Proceedings of the 8th International World Wide Web Conference*. W3C, Elsevier, May 1999.

[RSLC95] G. Rossi, D. Schwabe, C.J.P. Lucena, and D.D. Cowan. An Object-Oriented Model for Designing the Human-Computer Interface Of Hypermedia Applications. In *Proceedings of the International Workshop on Hypermedia design IWHD'95*, June 1995.

[Sch95] D. C. Schmidt. Using Design Patterns to Develop Reusable Object-Oriented Communication Software. *Communications of the ACM*, 38(10):65–74, October 1995.

[Sch99] D. Schwabe. Just add Water" Applications: Hypermedia Application Frameworks. In *2nd Workshop in Hypermedia Development: Design Patterns in Hypermedia*, 1999.

[SF89] P. D. Stotts and R. Furuta. Petri-Net-Based Hypertext: Document Structure with Browsing Semantics. *ACM Transactions on Information Systems*, 7(1):3–29, 1989.

[SF98] P. David Stotts and Richard Furata. Hyperdocuments as Automata: Verification of Trace-Based Browsing Properties by Model Checking. *ACM Transactions of Information Systems*, 16(1):1–30, 1998.

[SFC98] P. D. Stotts, R. Furuta, and C. Ruiz Cabarrus. Hyperdocuments as Automata: Verification of Trace-Based Browsing Properties by Model Checking. *ACM Transactions on Information Systems*, 16(1):1–30, 1998.

[SLHS93] J. L. Schnase, J. J. Leggett, D. L. Hicks, and Ron L. Szabo. Semantic Data Modeling of Hypermedia Associations. *ACM Transactions on Information Systems*, 11(1):27–50, January 1993.

[Spi92a] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 2nd edition, 1992.

[Spi92b] J.M. Spivey. *The fuzz Manual*. Computing Science Consultancy, 2 Willow Close, Garsington, Oxford OX9 9AN, UK, 2nd edition, 1992.

[SR94] D. Schwabe and G. Rossi. From Domain Models to Hypermedia Applications: an Object-Oriented Approach. In *Proceedings of the International Workshop on Hypermedia design IWHD'94*, 1994.

[SR95] D. Schwabe and G. Rossi. The Object-Oriented Hypermedia Design Model. *Communications of the ACM*, 38(8):45–46, August 1995.

[SRB95a] D. Schwabe, G. Rossi, and S. Barbosa. Abstraction, Composition and Lay-Out Definition Mechanisms in OOHDM. In *Proceedings of the ACM Workshop of Effective Abstractions in Multimedia*, 1995.

[SRB95b] D. Schwabe, G. Rossi, and S.D.J. Barbosa. Systematic Hypermedia Application Design with OOHDM. Technical Report 30,

Departamento de Informática, Pontifícia Universidade Católica, Rio de Janeiro, 1995.

[SRC95] L. F. G. Soares, N. L. R. Rodriguez, and M. A. Casanova. Nested Composite Nodes and Version Control in an Open Hypermedia System. *Information Systems*, 20(6):501–519, September 1995.

[SS99] J. Schummer and C. Schuckmann. Collaborative Hypermedia Design Patterns in OOHDM. In *2nd Workshop in Hypermedia Development: Design Patterns in Hypermedia*, 1999.

[SSGC98] C.A.S. Santos, L.F.G. Soares, G.L.Souza, and J. P. Courtiat. Design Methodology and Formal Validation of Hypermedia Documents. In *ACM Multimedia'98*, pages 39–48, 1998.

[TD96] K. Tochtermann and G. Dittrich. The Dortmund Family of Hypermedia Models – Concepts and their Application. *Journal of Universal Computer Science*, 2(1), 1996.

[Tom89] F. Wm. Tompa. A Data Model for Flexible Hypertext Database Systems. *ACM Trans. on Inf. Sys.*, 7(1):85, 1989.

[VCK96] J. M. Vlissides, J. O. Coplien, and N. L. Kerth, editors. *Patterns Languages of Program Design 2*. Addison-Wesley, 1996.

[vHL89] I. van Horebeek and J. Lewi. *Algebraic Specifications in Software Engineering*. Springer, 1989.

[Win90] J. M. Wing. A specifier's introduction to formal methods. *Compute*, 1990.

[WR98] W. Wang and R. Rada. Structured Hypertext with Domain Semantics. *ACM Transactions on Information Systems*, 16(4):372–412, 1998.

[YB00] J. Yoo and M. Bieber. Towards a Relationship Navigation Analysis. In *Proceedings of the 33rd Hawaii International Conference on System Sciences*, Jan. 2000.

Colophon

This dissertation was written and composed in its entirety using free software on two laptops: *Violeta*, a Compaq Presario 1210, with a Pentium 166 MHz and 32 Mbytes of memory, running Linux 2.0 (RedHat 5.0) and later Linux 2.2 (Linux 6.0); and *Iridium*, a Dell Inspiron 7500 with a Pentium III at 450 MHz and 192 Mbytes of memory, running Linux 2.2 (RedHat 6.1).

The typesetting was done in LaTeX2e, using tetex 1.0.6. The editing was done in emacs 20.4 usign AucTex 9.81. The diagrams were drawn using xfig 3.2 and PsTricks v97.

The main text is set in NewBaskervilleIT by Bitstream. The mathematics are set in Computer Modern by Donald Knuth.

The PostScript version was generated using dvips(k) 5.86 and later translated into Acrobat PDF by ghostscript 6.0.