Electronic Distribution of Searchable Technical Documentation Libraries

Elizabeth Martinez Aguilar

A Thesis

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science
Concordia University
Montréal, Québec, Canada

March 2003

0-612-77992-0

ABSTRACT

Electronic Distribution of Searchable Technical

Documentation Libraries

Elizabeth Martinez Aguilar

In this era of continuous evolution the publishing technology field has been constantly changing. The digital revolution of ten years ago brought such modifications that technological companies producing technical documentation had multiple difficulties learning new processes and integrating them into their publishing workflows. All that involved, and still involves today regular investments of time and money. Conversely, the beginning of the twenty first century brought an economic crisis that is affecting most of the telecommunications industry, forcing big reductions of expenses.

From the readers' perspective, the size of the technical documentation libraries keeps growing due to the increasing complexity of the subjects they discuss. While in the past paper documentation was the most in demand, nowadays the electronic format is the most popular since it can be manipulated in multiple ways. Readers need to locate and electronically extract the specific pages, paragraphs, graphics, or terms they are looking for. In addition, they need to navigate through thousands of pages without getting lost.

This thesis presents, through the use of the Unified Modeling Language (UML), the object-oriented design of a system that fulfills the demands of both the publishing companies and the users of their technical publications.

ACKNOWLEDGMENTS

I would like to thank my dear husband, Raymond, for his precious support through the development of this project. Also my gratitude goes to my loved children Alexandre, Guillaume, and Elizabeth whom represented a source of motivation towards the accomplishment of this important mission.

Special thanks to my supervisor, Dr. Gregory Butler, who always provided me with his invaluable guidance and support.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1 INTRODUCTION

1.1 The Problem

Companies selling high technology to their customers need to offer complete solutions in which technical documentation is an essential component. Because of the complexity involved in the installation, configuration, and ongoing maintenance of these high-tech products, the amount of the technical documentation can easily add up to the range of several thousand pages. While the printed format of this documentation is decreasing in demand, it is still a requirement that must be fulfilled; in contrast, the request for electronic documentation grows every day, and the Web, the medium most frequently used for its publication, does not always suit the customer's needs. Technicians in the field, who usually connect to the network through a telephone line, can not afford to wait for long minutes while the files download. In these cases, off-line access to the information is mandatory.

Customers want to receive the information in an easy to use, searchable format: in a library of about eighty books that could contain up to two thousand pages, they need to find quickly the book and specific page where the procedure covering a particular task is, for example, Installation of a Shelf. While performing that particular task, the technician might need to find an error message or an image explaining all the elements of the hardware component he is installing, or even more specifically the explanation of a technical term mentioned in that particular procedure. Moreover, if that procedure has a reference to another one in another book, the technician needs to navigate there and come back to the previous book afterwards. On other occasions, the users do not really know what they are looking for; they have a vague idea and only have in mind some technical terms; they would like to search for those terms in the whole library and see the book's titles and subjects where they appear.

To facilitate the learning process to their customers, high-tech companies are looking to have their technical documentation published in a consistent way and with an identical format regardless of the medium used: paper, CD-ROM, or the Web. They need to achieve this without sacrificing profitability. In many cases, they have acquired over the years a digital publishing apparatus focused on producing high quality printed documents, and based on the Adobe's Portable Document Format (PDF). Today, they must fulfill their customers' needs for on-line and off-line access to technical information, while facing reduced budgets, constant cuts of personnel, and increasingly tight schedules.

1.2 Background

Before PDF existed, the electronic exchange of files between print providers and their clients required all to have the same native applications and fonts that were used to create those files. Additionally, a paper proof of what the document should look like needed to be sent. Then around 1987, as Adobe's copyright states, a new file format for universal delivery, the Portable Document Format, was invented by Adobe. The publishing industry quickly adopted this new format for digital file interchange, since it offered the advantages of compact size and embedded fonts, images, and graphics that allowed having the same display and printing no matter the OS, monitor, or printer device. Their workflow greatly benefited from this new format and became more efficient, reliable, consistent, and economic.

The Adobe solution helped to streamline the publishing workflow which at the beginning was concentrated in the print version of the documents. PDF files offered three key advantages: reliability, consistency, and flexibility ["Adobe PDF"]. Their reliability ensured the document will have the same look on screen and on print, saving typical problems of missing fonts or graphics incorrectly displayed. Their consistency ensured they will look and behave the same way regardless of the OS platform or printing device used. Their flexibility allowed users to do proofing on both screen and paper, to make annotations and digital signatures, to extract pages, and to fully index and search, just to mention some of their features.

With the advent of the Web and its dramatic boost in popularity, the publishing industry was then confronted by the need for publishing on the Internet. HTML being the default format for the Web, technology evolved fast to provide it with exceptional viewing and search tools. At the present time the technology for searching PDF files is more limited than the one for searching HTML files ["Searching"]; although this represents a strong reason to change or adopt a new publishing format, it could be impossible to do because of a variety of reasons like a restricted budget, limited resources, or short deadlines.

From the manufacturing company perspective, a product complete solution must include technical documentation, training, and support. Technical documentation is in most cases written by technical writers, technical illustrators, and graphic designers and has to be developed in conjunction with the engineering and design teams. Although the documentation authoring activity is core to the company, security issues preventing its outsourcing, the documentation printing and distribution job is often given to an outside print specialist.

Technological products having a level of difficulty very high, their supporting documentation needs to include visual aids like headings, images, graphics, screens, and video in order to facilitate the customers' learning process. With all these visual aids the page counts of a book can increase very fast; at the same time, technology subjects are often very expansive. Readers do not like spending time trying to understand how the technical libraries are organized; the supporting documentation has to be very well planned so that end users can easily find what they are looking for in order to perform their work. Hence, technical writers need to show the book's summary and contents in a visual way easily assimilated. One way of achieving this is through the use of roadmaps that practically take the users by the hand through the task they need to accomplish.

Accordingly with the present e-business times, another customers' requirement is to receive the technical documentation in electronic format. To fulfill this requirement, one of the most common trends nowadays is to provide a Support Knowledge Web site where customers can find and download the electronic version of the documentation they need. However this option does not address all of the audience since a significant part are field technicians and installers who most of the time work in scarce conditions with only a laptop and a phone, without access to the Internet. In those cases, off-line access to the information is the only viable solution. Thus, the companies are now faced to the need for electronically publishing in different media, and want to provide an identical look and feel to their documentation regardless of the medium used for its distribution.

Printed documentation has been the traditional way to publish documentation. Over the years, technological companies equipped themselves to produce high quality printed documentation which in most cases was printed by a print provider. Consequently, they created publishing workflows to produce documents in a file format their print providers could handle. Currently, some of these print providers offer cross-media production services to deliver their customers' content to both print and the Web. Most of the time, they use third-party conversion software that can render documents created in a page layout application into the HTML format. However, these transformations bring on disadvantages, as Adobe maintains, "Unfortunately, such conversions often fail to render the designer's intentions on the Web. As with the workarounds in the print production workflow, the added steps are time-consuming and costly." ["Adobe PDF" 4].

The invention of the PDF format by Adobe provided a publishing solution that proved to be very efficient. PDF files are easy to deliver over the Web, across a network, or on removable media. They are compact and hold all the content unlike HTML files, which require you to include any linked graphics and font files.

1.3 Thesis Statement

In this e-business era, technological companies must adapt their traditional publishing system for the production and electronic delivery of technical documentation in a universal format that responds to their customers' demands for an easy to use, easy to install visual user interface that allows them to navigate and search through the numerous pages of the technical documentation libraries. Failure to do this would mean customers' dissatisfaction and loss of sales. On the other hand, these corporations are going through a prolonged economic crisis that forces them to keep their research and development budget reduced to a minimum. Even in these economic conditions, their publishing systems can still be modified to fulfill their customers' needs within very short deadlines and without sacrificing profitability.

1.4 My Work

I wanted to find a cost-effective, in-time deliverable solution for publishing technical libraries in an electronic format that could be universal, searchable, navigational, easy to learn, and easy to install. The research methodology I followed was interviews and surveys, as well as field experimentation through small prototypes. Close contact with the product engineers and designers, the technical writers, the technical illustrators, and the end customers was an essential part of this process.

My work targets companies that already have a print publishing solution based on the Adobe technology. The solution needed to be immediately applied to produce documentation on CD-ROM, and afterwards on the Web. Hence, I needed to design an application whose code could be reused.

Other essential characteristics of the solution should be:

- Enable customization of the visual interface so that it can be applied to different portfolios of products, while preserving a consistent look and feel.

- Automatically generated, so that production time is reduced to a minimum and the technical illustrators are better focused in producing the books' images instead of manually producing the visual interface for each product release.
- The automation tool to produce the visual interface will be operated by people with no programming background, most of the time the technical writers.

1.4.1 Research

While interviewing the technical writers, I found out they were concentrated in the writing process. Their work objectives were to get to know all of the product technical features and to acquire the necessary writing skills which included the use of Adobe Frame Maker, the tool used to produce PDF files. They found it more and more difficult to publish documentation on the World Wide Web, since their knowledge of JavaScript and other scripting languages was minimal.

The interviews with the engineers produced interesting results that were very useful for the design of the Graphical User Interface (GUI). These engineers are in close contact with the customers and constitute a channel for the latest to communicate their needs to the manufacturing company. Their work includes performing product validation tests, on-site training, phone support, and technical documentation review and approval. Their knowledge of programming languages is also very minimal.

The customer surveys showed the following:

- The technical documentation must be published in a consistent way, regardless of the medium used: paper, CD-ROM, or the Web.

- Users need to be able to manipulate the electronic document files: add comments, extract pages, make annotations, select text, select a graphic, etc.
- Users need to know what books they are entitled to receive when they buy a product solution.
- Users need to fully search the electronic files: search for a word in a book, in a subject, or in the whole library; search the text within a graphic or illustration; etc.
- The documentation must be navigable: books must be organized by product, platforms, general subject, and specific subject; and books must include internal links and cross-references to specific pages on other books.
- The documentation library must have a master index.

1.4.2 Choice of Technology

The choice of technology is limited by the following factors:

- The visual interface needs to support three major commercial platforms: Windows and UNIX Solaris and HP.
- The visual interface will be deployed in a first phase on CD-ROM and in a second phase on the World Wide Web.
- The technical documents need to be delivered in the PDF format.
- The automation tool that will generate the visual interface will be operated by people with no programming background.
- Very limited budget and tight schedules.

1.4.2.1 The Visual Interface

The solution needed to serve the companies currently using the Adobe workflows. This prerequisite reduced the spectrum of the available technology that I could use.

The Adobe technology

Adobe offers free software for viewing and printing PDF files, the Acrobat® Reader™. The distribution of this software via an intranet, a local network, a CD-ROM, or other physical media is authorized by Adobe upon an electronic license agreement. Besides its typical version which comes with a 'Find' command that reads the entire document when searching for a word or phrase, Acrobat Reader is also available as an expanded version on most platforms. This expanded version includes the 'Search' command that allows the user to search a collection of previously full-text indexed PDF books.

Acrobat Reader Search Tool

As explained on the Adobe Acrobat User Guide for Windows and Macintosh, opening a PDF document associated with an index automatically makes the index searchable. The user can search for a simple word or phrase, or he can expand his search by using Boolean operators and wild-card characters. He can refine or confine his search to documents listed in a prior search. For example, he can first search for all documents by a subject, and then define a search query for that subset of documents. The result would be a subset of documents belonging to that subject and that contain the search string. Additionally, he can use the options Proximity, Match Case, Sounds Like, Word Stemming, and Thesaurus on his search query. He can also select the documents to review from those returned by the search. If the PDF documents include 'Document Info' and

'Date Info', this information can be used in the Search dialog box, to limit the search. The Document Subject, Author, Title, Keywords, and Creation Date can be used in the search query.

Also from Adobe, the Adobe® Acrobat® software permits the conversion of any document to an Adobe Portable Document Format file. The PDF document can be open across a broad range of hardware and software, and it will look exactly as its author intended: with layout, fonts, links, and images intact. What is most interesting is the Acrobat Catalog feature of this software that allows the creation of full-text indexes of a single or of multiple Adobe PDF documents. Readers only need to search once to locate words or phrases in each of the documents sequentially.

PDF Files Navigation Features

Adobe PDF files can contain an assortment of navigation elements that makes it easier to locate specific information. Each Adobe PDF file automatically generates miniature previews of its pages called thumbnails, which can be used to navigate to a desired page instantly. In addition to thumbnails other navigational elements can be added or generated, including bookmarks and links. Bookmarks can serve as an interactive table of contents, which can be used to navigate to main topics instantly. Readers can use linked text and images to navigate immediately to a related page in the same document, in another Adobe PDF document, or in the World Wide Web.

Adobe Acrobat also allows the users to make annotations, add comments, extract or insert pages and graphics, and digitally sign the PDF documents.

Adobe® FrameMaker® and Adobe® FrameMaker®+SGML are the tools used by the authors to create the document source files. FrameMaker includes Acrobat®

Distiller™, for publishing to PDF format. The Distiller job settings permit the customization of the file compression, font embedding, color profiles, etc. When a FrameMaker document is saved as a PDF file, cross-references and hypertext commands automatically become links, and bookmarks can be automatically created. Metadata such as the author, title, subject, and keywords can be specified before it is exported to PDF. This information appears when the PDF file is printed and it is accessible to the Acrobat Catalog index and Acrobat Reader search command, and to some search engines for PDF files posted on the Web.

Other publishing software from Adobe that integrates seamlessly with FrameMaker are Adobe® Photoshop®, an image-editing application, and Adobe® Illustrator®, used to create vector graphics.

The Java Technology

The Java™ platform from Sun is based "on the power of networks and the idea that the same software should run on many different kinds of computers, consumer gadgets, and other devices. Since its initial commercial release in 1995, Java technology has grown in popularity and usage because of its true portability." [Hardee para.1]. The Java platform allows for the same Java application to run on multiple different kinds of computers. This is possible thanks to a component of the platform called the Java virtual machine (JVM™), a kind of translator that turns general Java platform instructions into tailored commands that make the devices do their work.

Sun has grouped its Java technologies into three editions: Standard Edition (J2SE™) refers to the general purpose version of Java 2 technology, Enterprise Edition (J2EE™) for large server based applications, and Micro Edition (J2ME™) for small devices such as cell phones. For the particular problem this thesis addresses, the most appropriate edition to be used was the Standard Edition which includes two principal components:

- The Java Runtime Environment (JRE) provides the libraries, Java virtual machine, the Java Plug-in and other components necessary to run applets and applications written in the Java programming language. It does not contain tools and utilities such as compilers or debuggers for developing applets and applications.
- The Java 2 Software Development Kit (J2SDK) is a superset of the JRE. It contains everything that is in the JRE, plus additional tools such as compilers and debuggers that are necessary for developing applets and applications.

Even when the Java virtual machine is incorporated into all major Web browsers, the built-in support in most of them uses an older version of Java technology, which cannot run applets using the most advanced features of the Java platform. Because of that it is more advantageous to install and use Sun's JVM that comes with the latest Java technology.

The Java Plug-in software extends the functionality of a Web browser allowing applets to be run under Sun's JRE rather than the Java runtime environment that comes with the Web browser. That means consistency and reliability when running applets. Java Plug-in is part of Sun's JRE and is installed with it. It works with both Netscape and Microsoft Internet Explorer and supports all Windows and most Solaris platforms. Sun has made Java Plug-in software available for porting to all operating system providers like Mac OS, AIX, Linux, and HP-UX. Starting with version 1.3, the Java Plug-in supports applets signed using Netscape signing tools. It can verify RSA signatures in a browser-independent way, through RSA and the .jar file. The new features in version 1.4 include support for multiple versions of the Java Runtime Environment installed on the same machine.

1.4.2.2 The Automation Tool

The automation tool will be used to generate automatically the visual interface, so that no special skills are required to produce it. In this way the use of time and resources is kept to a minimum, thus reflecting in the final cost of each CD-ROM.

<u>Microsoft Technology</u>

Microsoft technology is so widely spread that it has become a standard in the industry. Its visual tools allow for a very fast development of Windows based applications. Its programming language, Visual Basic (VB), makes it very easy to develop applications and to interact with other Microsoft products, like Excel or Word. The former comes with numerous pre-built components and interfaces; its tenth anniversary coincided with the release of Microsoft's new .NET platform and with a totally revised version, which is now fully object-oriented [Roman].

The automation tool will have a very simple visual interface, and will take its input from several Excel spreadsheets with a predefined template. As explained above, the choice of VB as a programming language will lead to a very fast development.

1.5 Contribution of the Thesis

This thesis presents the design of a cost-effective, fast-development automated solution for electronically delivering searchable, navigational technical documentation libraries on CD-ROM in a universal format that preserves an identical look of the information regardless of the publishing medium: paper, CD-ROM, or the Web. This solution fits into organizations having a pre-established publishing workflow based on the Adobe PDF format. Furthermore, the object-oriented design of the GUI is done so that its code can be reused afterwards for an implementation on the Web.

The complete solution includes the design of the graphical user interface as well as the design of the tool that automates its production. The graphical user interface can be customized for libraries of documents covering different products while preserving a consistent look and feel. The tool that produces the GUI is addressed to the technical writers who write the product technical documentation; consequently the knowledge of a programming or scripting language is not necessary.

The solution has the following valuable attributes:

- The GUI is robust and truly portable.
- The generation of the GUI is automated.
- The project's development time is measured in weeks.
- As per current economical standards, the total budget of the project is less than ten thousand dollars.

1.6 Thesis Organization

Chapter 2 presents the analysis of the problem via use cases and context diagrams. Chapter 3 presents the object-oriented design of the graphical user interface. Chapter 4 presents the design of the automation tool. Finally, chapter 5 presents a conclusion summarizing the results and possible improvements that could be made in the future.

CHAPTER 2 USE CASE VIEW OF THE SYSTEM

2.1 Modeling the Static Use Case View

This section presents the context of the system and its goals, from the end user's and analyst's point of view. The static use case view, which organizes the behaviors of the system, will be defined through the use of UML's use case diagrams and use case descriptions [Booch].

2.1.1 Scope and Goals

The Electronic Distribution of Searchable Technical Documentation Libraries system (EDSTDL) has two goals: the first one is to provide a visual interface to navigate easily through the books in a library and to search fully all the text and graphics in them; the second one is to automate the generation of such an interface, so that the resources dedicated to its production are kept to a minimum. Therefore, the system can be decomposed in two main components: the CD-ROM Graphical User Interface subsystem (CD-ROM GUI), and the CD-ROM Publishing subsystem (CD-ROM PUB).

The system's requirements and constraints are as follows:

- The GUI must be consistent through all the products in a portfolio.

- The initial distribution medium of the GUI is the CD-ROM, and the subsequent one is the Web.
- The GUI must run in Windows, Solaris-UX and HP-UX platforms.
- The format of the technical documents must be Adobe's PDF.
- The automation tool must be intended for people with no programming background.

2.1.2 The Context of the System

CD-ROM GUI Subsystem

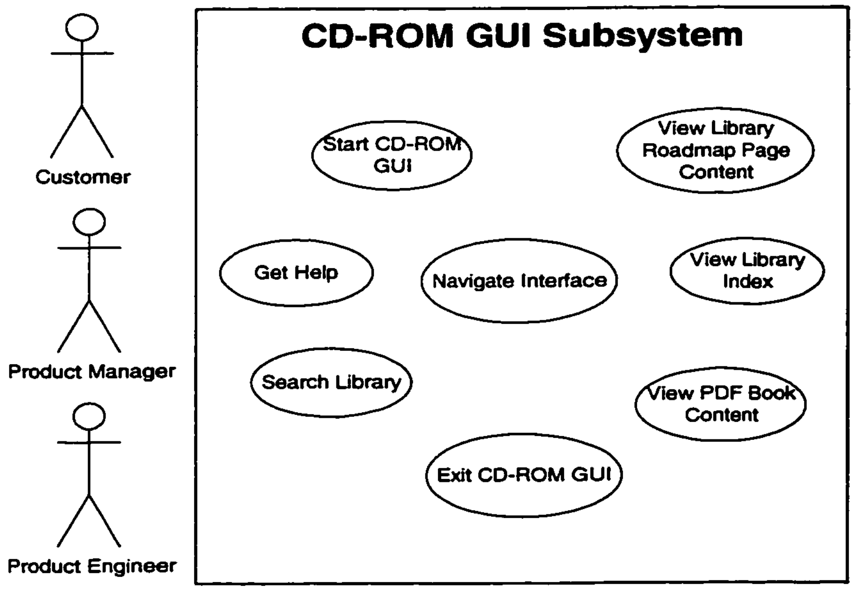


Figure 2.1 CD-ROM GUI Context Diagram.

Figure 2.1 above illustrates the actors who will be interacting with the CD-ROM GUI subsystem. The primary actors are the Customers who buy the company's products and technical documentation. As well, there are the Product Managers and Engineers who provide support to those customers.

The story to be supported is that the CD-ROM GUI user will insert the CD into his CD-ROM drive and the GUI will be launched automatically. The GUI should detect if all the necessary software elements are installed in the user's machine and prompt the user to install them if necessary. Once the installation process has finished and the user is interacting with the application, he must be able to see on a first screen the products, platforms, and releases that are included on the CD. After selecting one of them, he must be presented with a roadmap indicating the way to navigate through the books of that particular library section. When the Customer places the mouse over a particular book, its summary must be presented on a pop-up window. Once he clicks on a book, this one will be opened up. Within a book, there should be hyperlinks to figures, tables, and procedures, as well as a table of contents. If the book has references to an element in another book, a cross-reference link must be provided so that the Customer can go directly to the other book. As well, there should be a search tool within and outside the book that allows the user to make search queries. A master index to the library must be provided via a visual element like a button.

CD-ROM PUB Subsystem

As shown on Figure 2.2 on page 19, the primary actors of the CD-ROM PUB subsystem are the Technical Illustrator and the Technical Writer, who provide the input needed to create the visual interface. Another primary actor is the CD-ROM Operator, who burns the master CD and performs the tests and the verification of the final product.

Secondary actors involved in the GUI customization process are the Publishing and Product Managers.

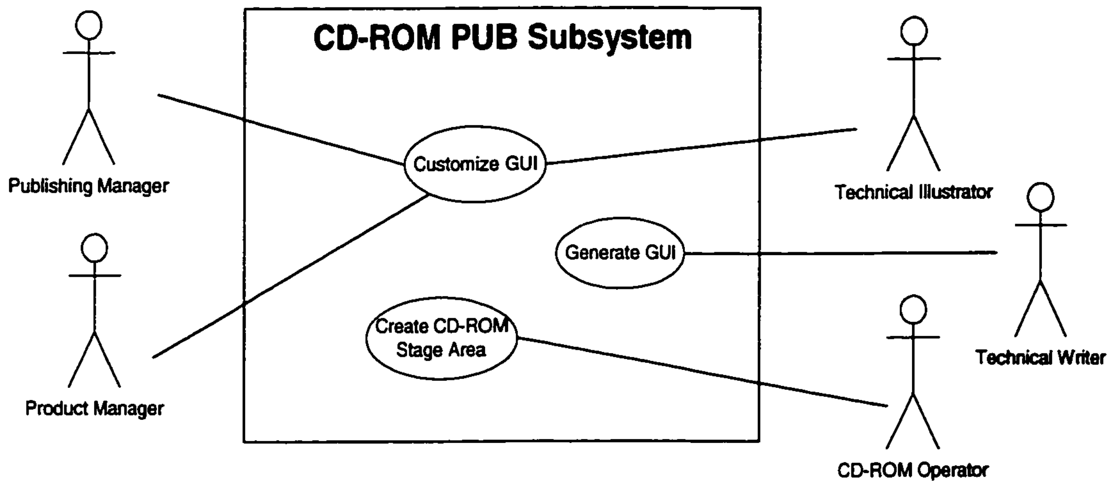


Figure 2.2 CD-ROM Publishing Context Diagram.

The story of the CD-ROM PUB subsystem begins when the Publishing and Product Managers have decided on the customization of the GUI for a particular portfolio: how many products to include and where should each of the product nodes be placed on the Network Map illustration. Once that decision has been taken, the Technical Illustrator creates the graphic for each product node included in the portfolio, and provides its exact position. He will enter those parameters into the CD-ROM PUB's Customization tool that will generate the template for the animation scripts. Once the GUI has been customized, the Technical Writer will be able to generate the GUI. He will input in an Excel file all the text needed: menu options which include products, platforms, and releases; roadmap column titles; book titles; pop-up text for each book; and paths to each book. The CD-ROM PUB's Generation tool will format all the input text into predefined lengths and will generate all the content pages. It will perform business rules validations and other validations like maximum text lengths for each screen element, screen specification missing, etc., and will generate a Log with processing errors and statistics. Finally, the CD-ROM Operator will create and validate the CD stage area using the CD-ROM PUB's Stage Creation tool. It will validate that all the GUI hyperlinks work correctly. Once that validation is correct and the stage area has been created, he will physically burn the master CD and will test it in all the required platforms. Once the CD-ROM GUI has passed all validation tests, he will create an ISO image of the master CD, scan it for virus and send it for replication to the CD-ROM distribution group.

2.1.3 Use Case Descriptions

CD-ROM GUI Subsystem

For this particular CD-ROM GUI subsystem, the actors are always the same: a Customer, a Product Engineer, a Product Manager, or any user using the CD-ROM.

1. Start CD-ROM GUI.

- *Goal in context:* Customer wants to see CD-ROM GUI.

- *Main success scenario:* Customer inserts CD into the CD-ROM drive of his computer. The CD-ROM automatically opens up the GUI's initial page containing the Network Map home page.

- *Extensions:* if user's computer is missing software required to run the GUI, an installation wizard screen pops up. User gives permission for the software to be installed. Once the installation finishes, the Network Map home page is displayed on the screen.

2. Navigate Interface.

- *Goal in context:* Customer wants to navigate through the GUI's content pages.

- *Main success scenario:* Customer selects one of the product nodes from the Network Map home page. Customer is presented with the Releases Menu and if available, with the Platforms Menu. He selects one of the menu options and is presented with the Library Roadmap page. From there, he can come back to the Network Map page by pressing the Back button.

- *Alternate scenario:* from the Network Map page or the Library Roadmap page, Customer can select the Help button that presents the corresponding Help screen. By pressing the Back button he can go back to the previous Library Roadmap page or to the Network Map page. By pressing the Back to Network Map page button he goes to the home page.

- *Alternate scenario:* from the Library Roadmap page, Customer selects the Search button; the Acrobat Reader helper application opens up, showing a PDF document explaining the use of the Acrobat Search tool. To go back to the previous screen, he simply puts the mouse over the Roadmap page window to activate it.

3. View Library Roadmap Page Content.

- *Goal in context:* Customer wants to view the content of one of the Library Roadmap pages.

- *Main success scenario:* includes Navigate Interface use case. Once Customer is presented with the Library Roadmap page, he puts the mouse over one of the buttons on the roadmap columns, and a pop-up box with the book summary is displayed. If a roadmap column is too long, the user can scroll up and down the roadmap column, one book at a time.

4. View PDF Book Content.

- *Goal in context:* Customer wants to see the contents of one of the books.

- *Main success scenario:* includes Navigate Interface use case. Once customer is presented with the Library Roadmap page, he places the mouse over one of the roadmap's book buttons, and clicks on it. The Acrobat Reader application is launched and opens up the corresponding PDF book.

5. View Library Index.

- *Goal in context*: Customer wants to see the index of the documentation library.

- *Main success scenario:* after selecting the desired product, release, and platform, Customer is presented with the Library Roadmap page. He clicks on the Index button and Acrobat Reader is launched and opens up the Index PDF book. If Customer clicks on the Index's cross-reference hyperlinks, he is taken to the corresponding book.

6. Search Library.

- *Goal in context:* Customer wants to search for something: a word, a term, a phrase, etc.

- *Main success scenario:* includes View PDF Book Content use case. Customer selects the Search icon on the Acrobat Reader's toolbar. Acrobat Reader displays the Search dialog box. Customer types in the search criteria and clicks on the OK button. Acrobat Reader presents a Search Results window with the list of hits. Customer double-clicks on one of the books listed and its contents are displayed on the screen.

- *Alternate scenario:* customer selects the Search button from the Library Roadmap page. Acrobat Reader is launched and opens up a PDF document showing instructions on how to use the Search feature. Customer clicks on the Acrobat Reader's Search icon and Acrobat Reader displays the Search dialog box.

7. Get Help.

- *Goal in context:* Customer needs help on using one of the GUI's features.

- *Main success scenario:* Customer clicks on the Help button on one of the GUI's screens. He is presented with a Help page explaining each of the GUI's features. User clicks on one of the feature hyperlinks and he is taken to the corresponding section.

8. Exit CD-ROM GUI.

- *Goal in context:* Customer wants to close the CD-ROM GUI application.

- *Main success scenario:* Customer clicks on the Exit button on one of the GUI's screens. An Exit confirmation window appears and user clicks on the OK button. CD-ROM GUI is closed.

- *Alternate scenario:* Customer clicks on the Close icon at the right top corner of the browser window. An Exit confirmation window appears and user clicks on the OK button. CD-ROM GUI is closed.

<u>CD-ROM PUB Subsystem</u>

1. Customize GUI.

- *Goal in context:* Technical Illustrator wants to perform the customization of the GUI, so that it reflects a new portfolio of products.

- *Actors:* Technical Illustrator.

- *Main success scenario:* Technical Illustrator creates the portfolio artwork accordingly to the Publishing Manager's specifications. Once he finishes, he starts the Customization tool from the CD-ROM PUB system. He is presented with the Customization screen where he enters the required information: portfolio artwork location, base template location, and output location. He clicks on the OK button. A status bar appears indicating the progress of the GUI Customization process. When the process is finished, a message appears indicating the status of the operation as well as the complete path where the output files were saved.

2. Generate GUI.

- *Goal in context:* Technical Writer wants to generate the GUI's content pages.

- *Actors:* Technical Writer.

- *Main success scenario:* Technical Writer prepares an Excel file with the information about the product, release and platform menu options, book summary pop-up boxes, roadmap columns, book titles, and file names. After that, he starts the Generation tool from the CD-ROM PUB system. He enters the input parameters such as specification file location, Network Map template location, and the output location. He clicks on the OK button. A status bar appears indicating the progress of the Generation process. When the process is finished, a message appears indicating the status of the operation as well as the complete path where the output files were saved.

3. Create CD-ROM Stage Area.

- *Goal in context:* CD-ROM Operator wants to validate and generate the CD-ROM stage area.

- *Actors:* CD-ROM Operator.

- *Main success scenario:* CD-ROM Operator prepares the Acrobat index files needed for each of the product options. He prepares as well other input files needed like the Release Notes and the Search User Guide files. After that, he selects the Stage Creation tool from the CD-ROM PUB system. He is presented with the Stage Creation screen where he types in the required information: location of the directories where content pages, PDF files, and other input files are located; and root directory where CD-ROM virtual partition is located. Once he finishes entering the input parameters, he clicks on the OK button. A status bar appears indicating the progress of the Stage Creation process. When the process is finished, a message appears indicating the status of the operation as well as the complete path where the output files were saved.

2.2 Modeling the Structure: Class Diagrams

After having defined the requirements of the system and its principal actors, we can now start modeling the vocabulary of the system. This is done by deciding on which of the previously defined abstractions fall into the system's boundary. Similarly, we can model the associations and collaborations between those abstractions via class relationships.

CD-ROM GUI Subsystem

Figure 2.3 on page 26 shows the domain model for the CD-ROM GUI subsystem. This model presents the conceptual classes and associations most representative of the problem. Since one of the system's future requirements is its deployment on the World Wide Web, the GUI is browser-based. The GUI's home page is represented by the *NetworkMapPage* class that contains an introductory text and a picture showing a network of product nodes represented by the *ProductNode* class. The nodes corresponding to the products included in a particular CD-ROM appear animated. The *NetworkMapPage* class also contains the *ReleasesMenu* and *PlatformsMenu* classes as well as a link to a Help page which is represented by the *HelpPage* class. The subsequent pages correspond to each of the Library Roadmaps and are illustrated by the *RoadmapPage* class. Each of these pages

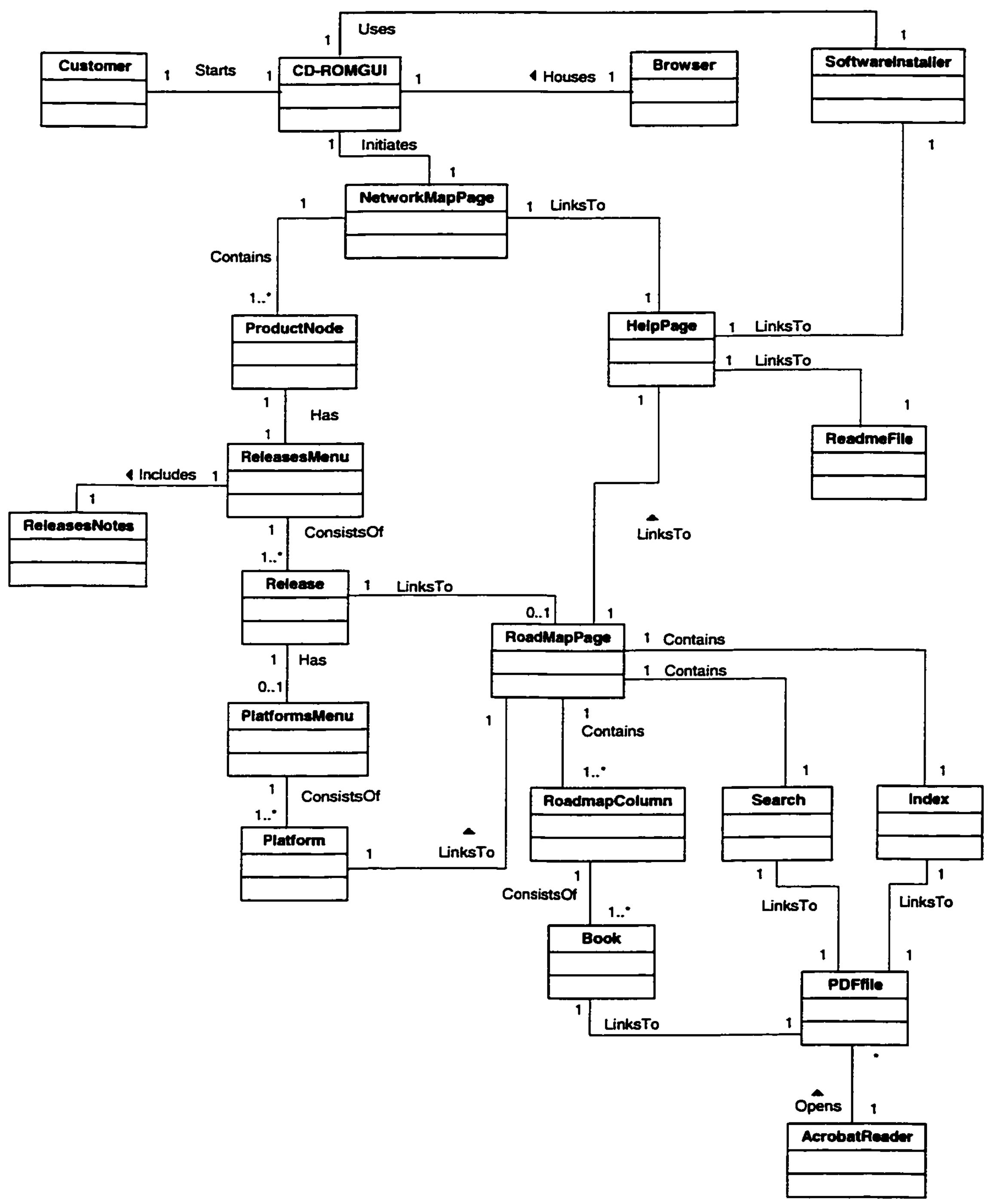


Figure 2.3 Domain Model for the CD-ROM GUI Subsystem.

contains links for the following: PDF technical documents, PDF Search User Guide, PDF Library Master Index, and Help page.

CD-ROM PUB Subsystem

Figure 2.4 on next page illustrates the domain model for the CD-ROM PUB subsystem. This model presents the conceptual classes and associations that help understand the system's functionalities needed by the different users. Some of these classes, like the *ProductManager* and the *PublishingManager*, might end up not appearing in the final object model however, they are presented here because they indirectly affect the system. The CD-ROM PUB system offers its services through the classes: *GUICustomization*, *GUIGenerator*, and *StageCreator*. Each of these services is initiated by the corresponding primary actors represented by the classes: *TechnicalIllustrator*, *TechnicalWriter*, or *CD-ROMOperator*. These actors produce before hand the input needed by the publishing system: Portfolio Artwork, GUI Specification, Acrobat Index files, Acrobat PDF files, etc. There is a class common to all of the services classes, the *Validator* class that is in charge of performing the business rules validations and that produces a log of events and errors represented by the *Log* class. Although the service classes encapsulate most of the system functionalities there is one, the *LinkChecker* class, which is specified separately since it is an essential feature the system provides. Similarly, the classes *AcrobatIndex* and *PDFfile* help identify one of the external systems our subsystem interacts with. The final output of the CD-ROM PUB subsystem is represented by the *VirtualCD-ROM* class that contains the classes *NetworkMapPage*, *RoadmapPage*, *PDFfile*, and *AcrobatIndex*. An intermediary output would be the *NetworkMapTemplate* class that is needed for the GUI Generation service.

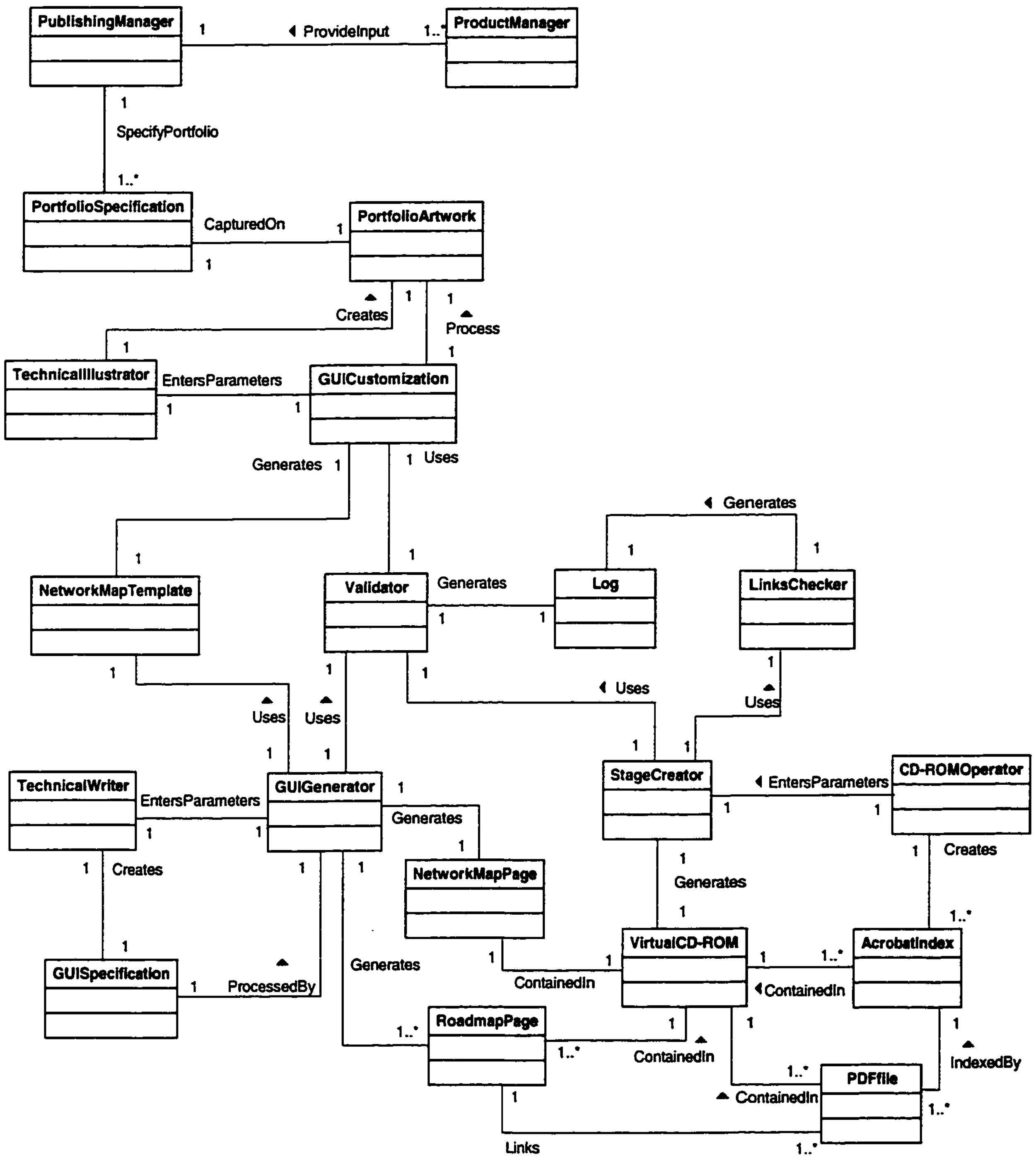


Figure 2.4 Domain Model for the CD-ROM Publishing Subsystem.

2.3 Modeling the Behavior

The system behaviors were previously defined via the static use case definitions. As well, in the previous section we defined the system domain classes and their collaborations. We can now model the dynamic aspect of those collaborations by using interactions. "An interaction is a behavior that comprises a set of messages exchanged among a set of objects within a context to accomplish a purpose." [Booch 205]. These interactions can be modeled by emphasizing the time ordering of messages via Interaction Diagrams, or by emphasizing the flow of control from activity to activity via Activity Diagrams.

2.3.1 Activity Diagrams

Figure 2.5 on page 30 shows the Activity Diagram for the use case Search Library, belonging to the CD-ROM GUI subsystem. To visualize better the whole operation of the system, the diagram includes the tasks associated to the Start CD-ROM GUI and Navigate Interface use cases. The application is intelligent enough to detect if software is missing and practically takes the user by the hand through the installation steps. Once the customer clicks on a roadmap button to open a book, the GUI application launches the Acrobat Reader subsystem. Within Acrobat Reader, the customer invokes the Search tool and types in his search criteria. The search results are displayed on the screen.

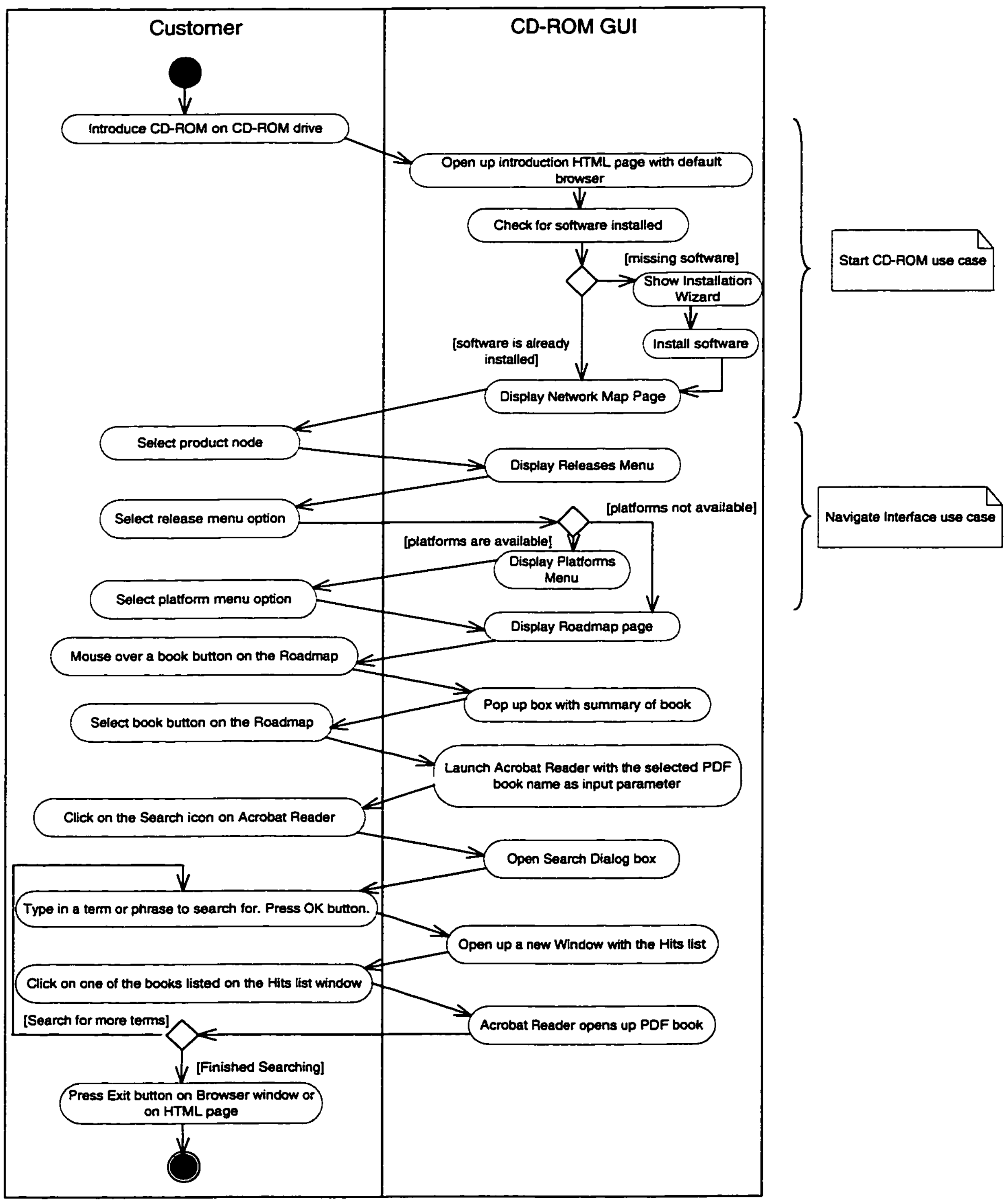
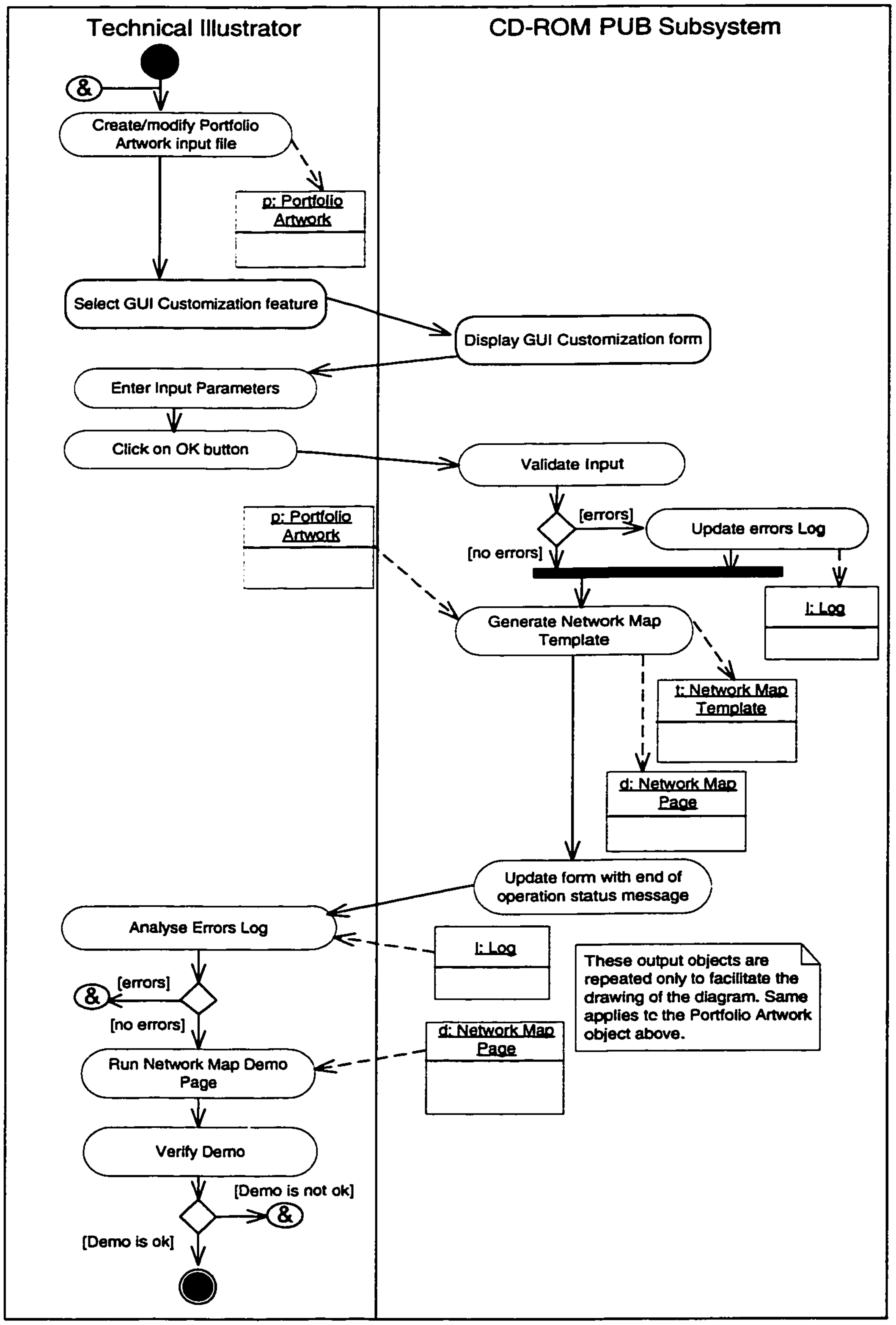

Customer
CD-ROM GUI
Introduce CD-ROM on CD-ROM drive
Open up introduction HTML page with default browser
Check for software installed
[missing software]
Show Installation Wizard
Install software
[software is already installed]
Display Network Map Page
Start CD-ROM use case
Select product node
Display Releases Menu
Select release menu option
[platforms not available]
[platforms are available]
Display Platforms Menu
Navigate Interface use case
Select platform menu option
Display Roadmap page
Mouse over a book button on the Roadmap
Pop up box with summary of book
Select book button on the Roadmap
Launch Acrobat Reader with the selected PDF book name as input parameter
Click on the Search icon on Acrobat Reader
Open Search Dialog box
Type in a term or phrase to search for. Press OK button.
Open up a new Window with the Hits list
Click on one of the books listed on the Hits list window
Acrobat Reader opens up PDF book
[Search for more terms]
[Finished Searching]
Press Exit button on Browser window or on HTML page

Figure 2.5 Activity Diagram for Search Library Use Case.



Technical Illustrator
CD-ROM PUB Subsystem
&
Create/modify Portfolio Artwork input file
p: Portfolio Artwork
Select GUI Customization feature
Display GUI Customization form
Enter Input Parameters
Click on OK button
Validate Input
[errors]
Update errors Log
[no errors]
p: Portfolio Artwork
l: Log
Generate Network Map Template
t: Network Map Template
d: Network Map Page
Update form with end of operation status message
Analyse Errors Log
l: Log
These output objects are repeated only to facilitate the drawing of the diagram. Same applies tc the Portfolio Artwork object above.
[errors]
[no errors]
d: Network Map Page
Run Network Map Demo Page
Verify Demo
[Demo is not ok]
[Demo is ok]

Figure 2.6 Activity Diagram for Customize GUI Use Case.

Figure 2.6 on previous page, presents the activity diagram for the use case Customize GUI performed by the Technical Illustrator and belonging to the CD-ROM Publishing subsystem. We can see the interactions between the Technical Illustrator human subsystem and the CD-ROM Publishing subsystem. While the Technical Illustrator is the one who prepares the artwork, the GUI Customization module of the publishing system takes care of processing it and generating the Network Map template. A demo that consists of the Network Map page with all products nodes animated is also generated. This helps the illustrator to perform his verifications. The Network Map template is an input element for the GUI Generation use case.

2.3.2 System Sequence Diagrams

In the following diagrams, the subsystems we are modeling appear as a whole and are represented by a system class. The details inside each of these subsystems will be explained in the next two chapters. Notice that user's input is also represented as separate classes since it fits into the present use case view context.

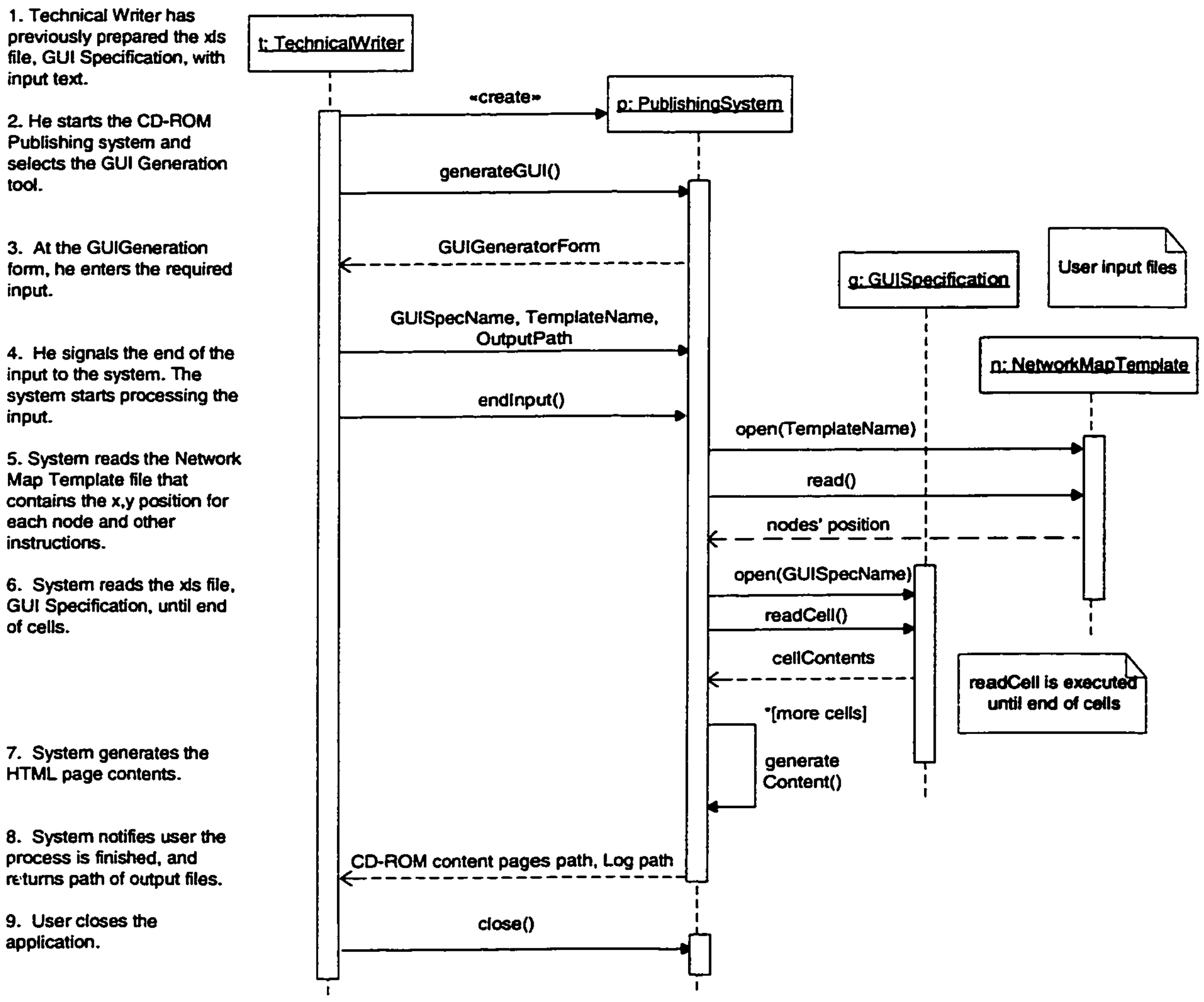

1. Technical Writer has previously prepared the xls file, GUI Specification, with input text.
2. He starts the CD-ROM Publishing system and selects the GUI Generation tool.
3. At the GUIGeneration form, he enters the required input.
4. He signals the end of the input to the system. The system starts processing the input.
5. System reads the Network Map Template file that contains the x,y position for each node and other instructions.
6. System reads the xls file, GUI Specification, until end of cells.
7. System generates the HTML page contents.
8. System notifies user the process is finished, and returns path of output files.
9. User closes the application.
t: TechnicalWriter
p: PublishingSystem
g: GUISpecification
User input files
n: NetworkMapTemplate
«create»
generateGUI()
GUIGeneratorForm
GUISpecName, TemplateName, OutputPath
endInput()
open(TemplateName)
read()
nodes' position
open(GUISpecName)
readCell()
cellContents
*[more cells]
readCell is executed until end of cells
generate Content()
CD-ROM content pages path, Log path
close()

Figure 2.7 Sequence Diagram for Generate GUI Use Case.

The Sequence Diagram on Figure 2.7 above, illustrates the use case Generate GUI. We can identify some of the operations the Publishing System, as a black box, offers in its public interface to handle incoming system events. The system event *generateGUI* invokes a system operation *generateGUI* and the system event *endInput* invokes a system operation *endInput*. As stated by Craig Larman, "The entire set of system operations, across all use cases, defines the public system interface." [Larman 178].

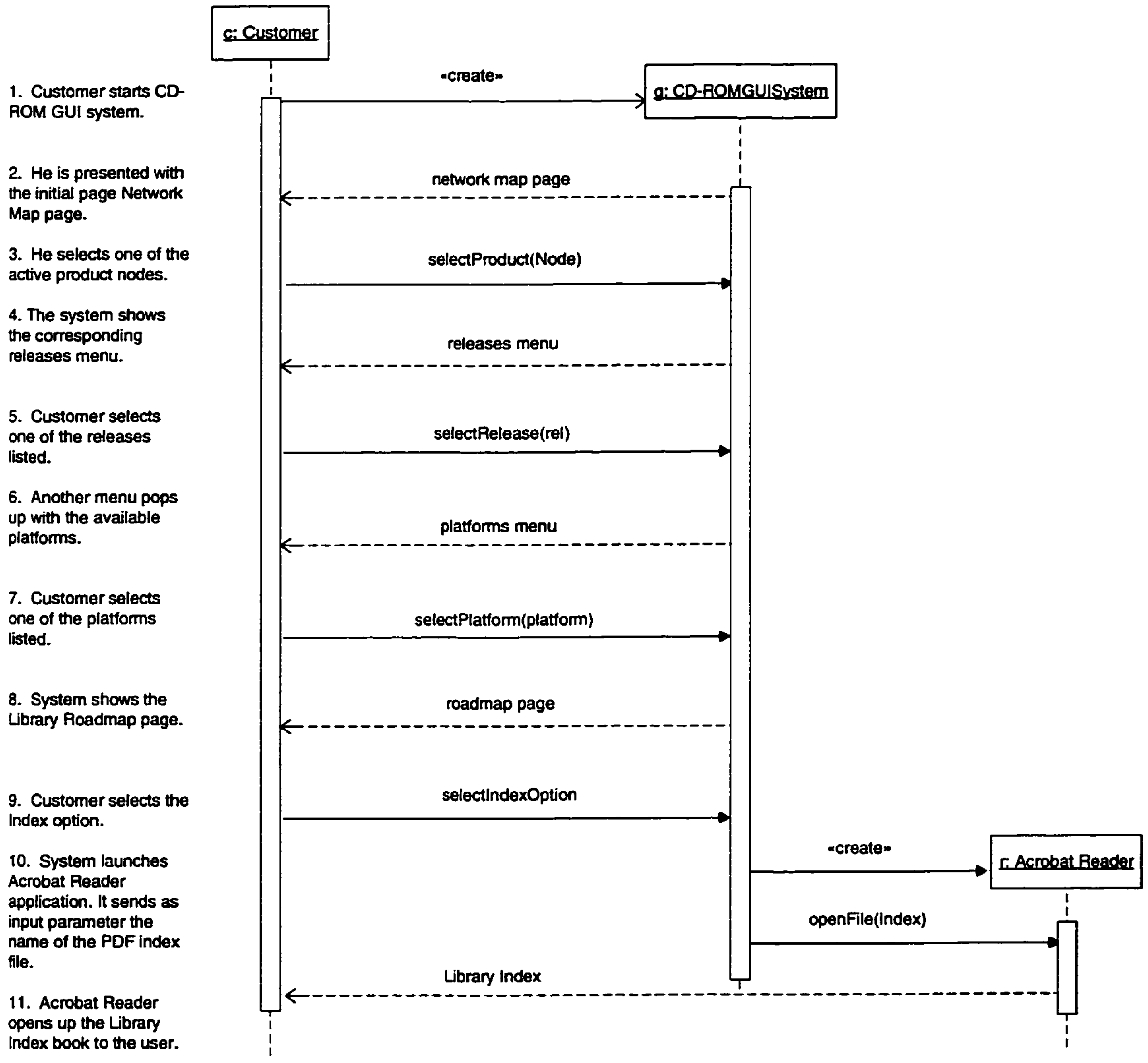

c: Customer
«create»
g: CD-ROMGUISystem
1. Customer starts CD-ROM GUI system.
2. He is presented with the initial page Network Map page.
network map page
3. He selects one of the active product nodes.
selectProduct(Node)
4. The system shows the corresponding releases menu.
releases menu
5. Customer selects one of the releases listed.
selectRelease(rel)
6. Another menu pops up with the available platforms.
platforms menu
7. Customer selects one of the platforms listed.
selectPlatform(platform)
8. System shows the Library Roadmap page.
roadmap page
9. Customer selects the Index option.
selectIndexOption
10. System launches Acrobat Reader application. It sends as input parameter the name of the PDF Index file.
«create»
r: Acrobat Reader
openFile(Index)
11. Acrobat Reader opens up the Library Index book to the user.
Library Index

Figure 2.8 Sequence Diagram for View Library Index Use Case.

Figure 2.8 above shows the Sequence Diagram for the View Library Index use case. An interesting thing to note is the interaction of the CD-ROM GUI subsystem with an external system, the Acrobat Reader application. We will see in Chapter 3 exactly how Acrobat Reader is being invoked.

2.4 Subsystems Interdependencies

The two components that make up the EDSTDL system are closely related.

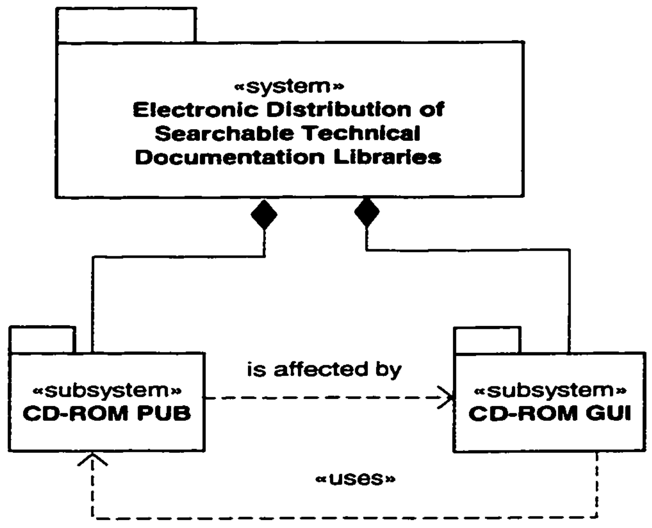


Figure 2.9 System Decomposition into Subsystems.

We can see on Figure 2.9 above that there is a using relationship from the CD-ROM GUI subsystem to the CD-ROM PUB subsystem; this is due to the fact that the publishing subsystem produces the variable elements that make up the GUI. There is also a dependency relationship from the CD-ROM Publishing subsystem to the CD-ROM GUI subsystem because modifications to the business rules might cause the CD-ROM GUI implementation to be changed, and consequently the CD-ROM PUB subsystem would need to be adapted to reflect those changes.

2.5 Recapitulation

The requirements, concepts, and operations related to the two subsystems CD-ROM Graphical User Interface and CD-ROM Publishing have been presented in this chapter. The domain vocabulary has been defined as well as the principal system goals, related rules and constraints, and interdependencies. We have learned what needs to be done in order to satisfy the end users' needs. We are now ready to move from the requirements phase to the design and implementation phases.

CHAPTER 3 CD-ROM GUI DESIGN

3.1 Design Rationale

The CD-ROM GUI system is a browser-based, interactive application used to view and search technical documentation. Its aim is to provide the user with exceptional navigational tools and search features so that he makes the most of his learning experience. The user interacts with the system via a graphical user interface which has to be highly usable. User interaction with the system is done via mouse clicks and only when using the Adobe search feature, via the keyboard; validation of user input is thus practically nonexistent in this system. Moreover, the information contained in the visual interface has already been validated and tested; unless there are unexpected changes to the software layers and other subsystems our application interacts with--OS, browser, Java virtual machine, or Adobe Acrobat Reader--application's logic errors should not happen, only hardware failure kind of errors can occur.

Design Patterns

Frank Buschmann states when talking about interactive systems: "When specifying the architecture of such systems, the challenge is to keep the functional core independent of the user interface." [Buschmann 123]. In the case of the CD-ROM GUI, the functional core is represented by the business rules dictating the standards for writing the technical documentation and by the data model underlying the application; these functional requirements are not likely to change very often. On the other hand, although the presentation of the visual user interface has been carefully analyzed and its conception is the result of extensive studies and tests with the end users, it is possible that it will need to be changed due to customer's needs arising and evolving quite often. Consequently, in order to provide the system with the highest flexibility for future extensions and maintenance, the application's functionality needs to be separated from its graphical user interface. This can be achieved by applying the Model-View-Controller pattern (MVC), a classic design pattern for interactive applications that divides them into three areas: processing, output, and input. The design of the CD-ROM GUI was inspired by the MVC, although this pattern was not applied in a strict manner.

As explained in the previous chapter, there is a dependency relationship between this CD-ROM GUI subsystem and its counterpart, the CD-ROM PUB. The separation of the Model and the View components will help to the better control of this dependency relationship: changes to the presentation layer of the CD-ROM GUI will not affect the model, therefore the CD-ROM PUB, the one that generates the model, will not be affected.

Technology

The technology for enabling browser-based client applications relies on the Document Object Model (DOM), a platform-neutral interface to the browser and the documents it renders. DOM specification has been defined by the World Wide Web Consortium (W3C) and most browser manufacturers have implemented it. With the DOM, programs and scripts can dynamically access and update the content, structure and style of documents [Conallen].

JavaScript, which is the most common scripting technology in browsers today, is an easy yet powerful mechanism to use. Since it is built into the browser, it is fast to load. JavaScript is object based, not object-oriented; it uses the objects provided by the DOM, but it also allows the creation of custom objects by the user. Finally, Dynamic HTML (DHTML) is a term used to describe the combination of HTML, style sheets, and scripts that allow page authors far more control over the display and interactive behavior of Web page content.

The Sun's Java technology previously described in the second chapter is also being used, specifically in the Roadmap page implementation.

Design Organization

The CD-ROM GUI application has two kinds of static, animated pages: the Network Map home page and the Roadmap pages. This chapter will present the design of each of these pages. The decisions taken on which domain classes to keep, which classes to make up, the corresponding assignment of responsibilities, and the design patterns to apply, are all illustrated via UML's class and interaction diagrams.

3.2 Network Map Page Design

The Network Map page is responsible for displaying the network of product nodes, differentiating via some kind of animation the nodes that are active in a particular CD-ROM. When selected, each active node must display the corresponding releases and platforms menus. When one of the menu options is clicked, it should link to the corresponding Roadmap page. Figures 3.1 on next page shows a snapshot of the Network Map page.

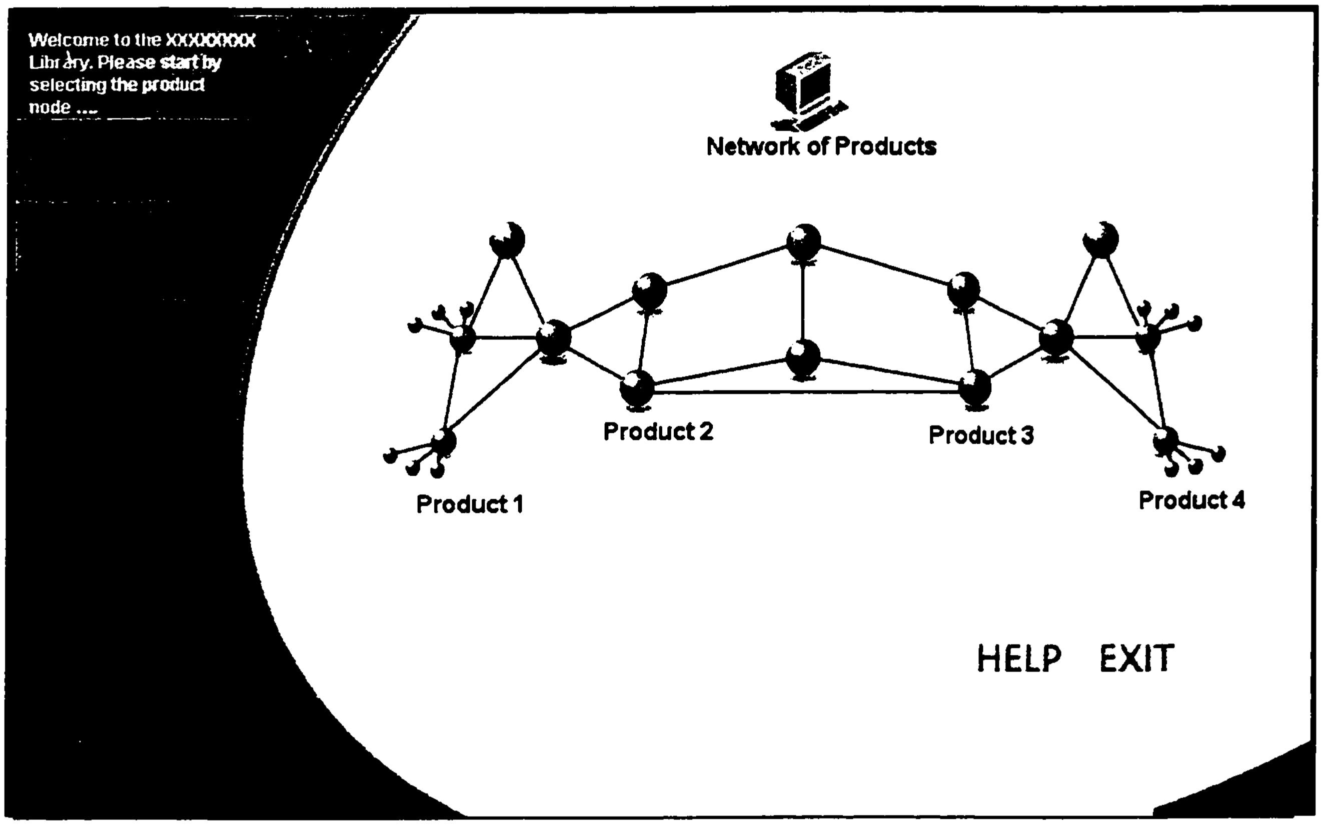

Welcome to the XXXXXXXX Library. Please start by selecting the product node ....
Network of Products
Product 1
Product 2
Product 3
Product 4
HELP
EXIT

Figure 3.1 Network Map Page.

DHTML is being used to build the Network Map page. Some DOM objects being employed are: *document*, *image*, *anchor*, *link*, and *layer*. As well, some custom objects are created: *Menu*, *MenuItem*, *ActiveNode*, *WelcomeText*, and *Button*. JavaScript is being used to handle the clicks and other mouse events. Even when these functions are not architecturally important, they are shown here since they are essential to the user interface model.

3.2.1 Behavioral Model: Sequence and Activity Diagrams

Figure 3.2 on page 42 shows the sequence diagram for the use case Start CD-ROM GUI, which basically consists on the browser presenting the Network Map page to the user. The browser is responsible for rendering the HTML document, and its built-in JavaScript interpreter executes the scripts. The JavaScript functions are organized into a utility library, the Layers.js file, represented in the diagram by the *Layers* object. This object contains the function definitions that encapsulate the behaviors needed to handle the system events that occur when the user interacts with the application. Therefore, all the functions and custom object definitions included in the Layers.js utility library represent a Controller component. The Model is partially represented by the nodes' constant positions defined in the Layers.js file, as well as by the Loader.js file, shown in the diagram as the *Loader* object, which contains calls--with constant values as parameters--to the functions previously defined in the Layers.js file. These calls execute only the code needed when the document is first loaded. Afterwards, when the user interacts with the browser and an event is fired via a DOM object, it will be handled by the JavaScript function whose name is indicated on the eventhandler parameter of the DOM object. The browser, which implements the DOM interface, delegates the event handling to the JavaScript functions defined in the Layers.js file. Hence, the browser is basically acting as a View component; it is not handling the events.

On Figure 3.2, we can see that after the user starts the CD-ROM GUI application, the browser is either launched automatically in the case of a Windows platform, or manually in the case of UNIX. The browser retrieves the Network Map HTML page from

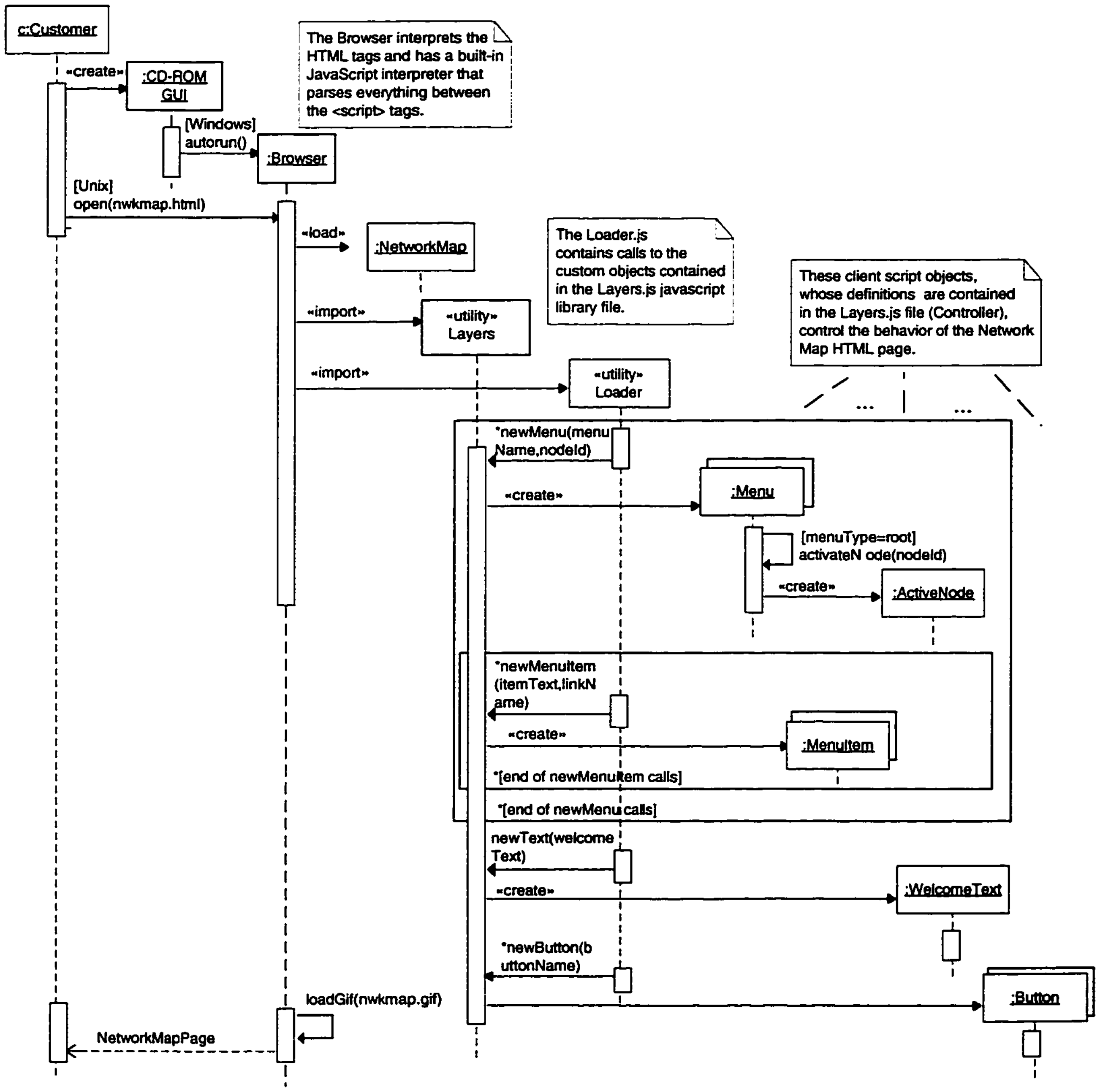
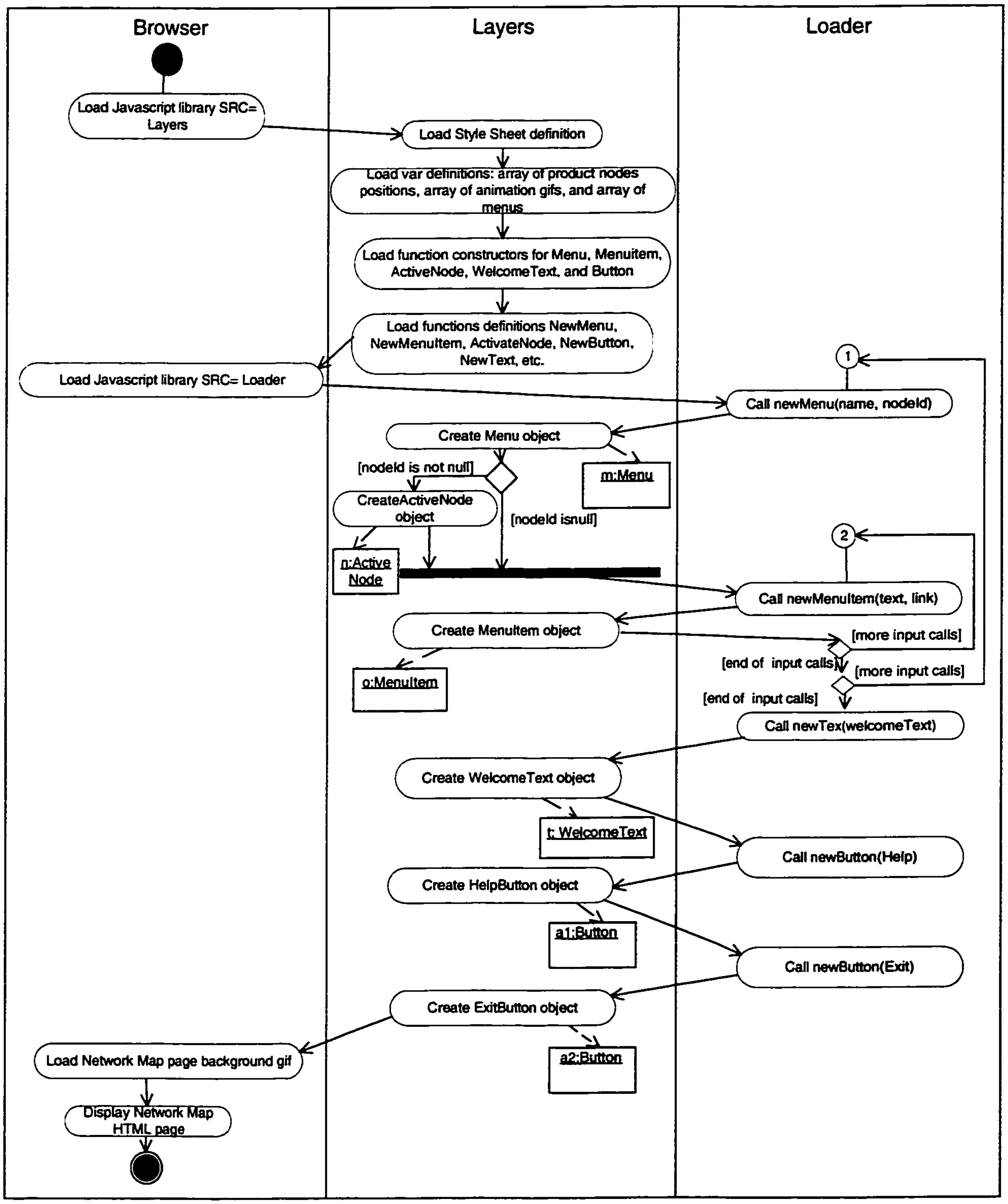

c:Customer
«create»
:CD-ROM GUI
[Windows] autorun()
:Browser
The Browser interprets the HTML tags and has a built-in JavaScript interpreter that parses everything between the <script> tags.
[Unix] open(nwkmap.html)
«load»
:NetworkMap
«import»
«utility» Layers
The Loader.js contains calls to the custom objects contained in the Layers.js javascript library file.
«import»
«utility» Loader
These client script objects, whose definitions are contained in the Layers.js file (Controller), control the behavior of the Network Map HTML page.
*newMenu(menuName,nodeId)
«create»
:Menu
[menuType=root] activateNode(nodeId)
«create»
:ActiveNode
*newMenuItem (itemText,linkName)
«create»
:MenuItem
*[end of newMenuItem calls]
*[end of newMenu calls]
newText(welcomeText)
«create»
:WelcomeText
*newButton(buttonName)
:Button
loadGif(nwkmap.gif)
NetworkMapPage

Figure 3.2 Sequence Diagram for Start CD-ROM GUI Use Case.

the CD-ROM drive location and starts loading it. While parsing the HTML tags, it detects the Script tags and imports the JavaScript utility library represented by the *Layers* object. The style sheet definitions, variables and constants declarations, utility functions, and custom object's prototypes--constructors--are all loaded. After that, the Loader.js file represented by the *Loader* object is imported as well; it contains calls to the Layer's functions previously loaded. The function calls *newMenu* instantiate one *Menu* custom object for each Releases and Platforms menu needed. After that, the function calls *newMenuItem* instantiate the collection of *MenuOption* objects that are contained in each of the *Menu* objects. The call to the *newText* function instantiates the *WelcomeText* object, which basically is a layer showing some welcome text. The calls to the *newButton* functions, instantiate the different *Button* objects. Finally, after the Loader's JavaScript interpretation finishes, the HTML body tags of the document are interpreted by the browser. It loads the background image and renders the document to the user.

Figure 3.3 on page 44 illustrates the flow of activities performed when the Network Map HTML page is being loaded by the browser, which is a task performed within the Start CD-ROM GUI use case. The browser's built in JavaScript engine interprets the instructions enclosed in the imported JavaScript libraries Layers.js and Loader.js. These two physical components appear here as kind of subsystems. Once all the JavaScript custom objects are created, the control comes back to the browser who then finishes interpreting all the HTML tags in the document. In our case, the background element of the *TD* tag indicates to the browser that a *gif* must be loaded. The end of the HTML document is detected and the browser renders the page into the screen.



Browser
Layers
Loader
Load Javascript library SRC= Layers
Load Style Sheet definition
Load var definitions: array of product nodes positions, array of animation gifs, and array of menus
Load function constructors for Menu, MenuItem, ActiveNode, WelcomeText, and Button
Load functions definitions NewMenu, NewMenuItem, ActivateNode, NewButton, NewText, etc.
Load Javascript library SRC= Loader
1
Call newMenu(name, nodeId)
Create Menu object
m:Menu
[nodeId is not null]
CreateActiveNode object
[nodeId isnull]
n:Active Node
2
Call newMenuItem(text, link)
Create MenuItem object
[more input calls]
[end of input calls]
[more input calls]
o:MenuItem
[end of input calls]
Call newTex(welcomeText)
Create WelcomeText object
t: WelcomeText
Call newButton(Help)
Create HelpButton object
a1:Button
Call newButton(Exit)
Create ExitButton object
a2:Button
Load Network Map page background gif
Display Network Map HTML page

Figure 3.3 Activity Diagram for Network Map Page Loading Process.

3.2.3 Structural Model: Class Diagram

Figure 3.4 on page 46 shows the class diagram for the Network Map HTML page. Its focus is in presenting the JavaScript custom objects that get created during the Start CD-ROM GUI use case. We are showing only the variables that are global and therefore visible to the Network Map page itself. The *SCRIPT* elements in the HTML page have a *SRC* attribute with the values Layers and Loader. This is represented in the class diagram via the stereotyped «script library» classes. A «script» stereotyped association between the «client HTML page» *NetworkMap* class and the «script library» *Loader* and *Layers* classes is shown. All JavaScript var and function definitions are forward engineered into the «script library» classes, not the client page [Conallen].

The *Layer* object provided by the DOM is being used by most of the custom script objects that provide animation. A layer is a container that is capable of holding a HTML document and exists in a plane in front of the main document. We are moving, sizing, and hiding the layer by the means of JavaScript methods provided by the custom objects. The *Link* and *Image* DOM objects are also used by the custom objects even though they are not shown in the diagram.

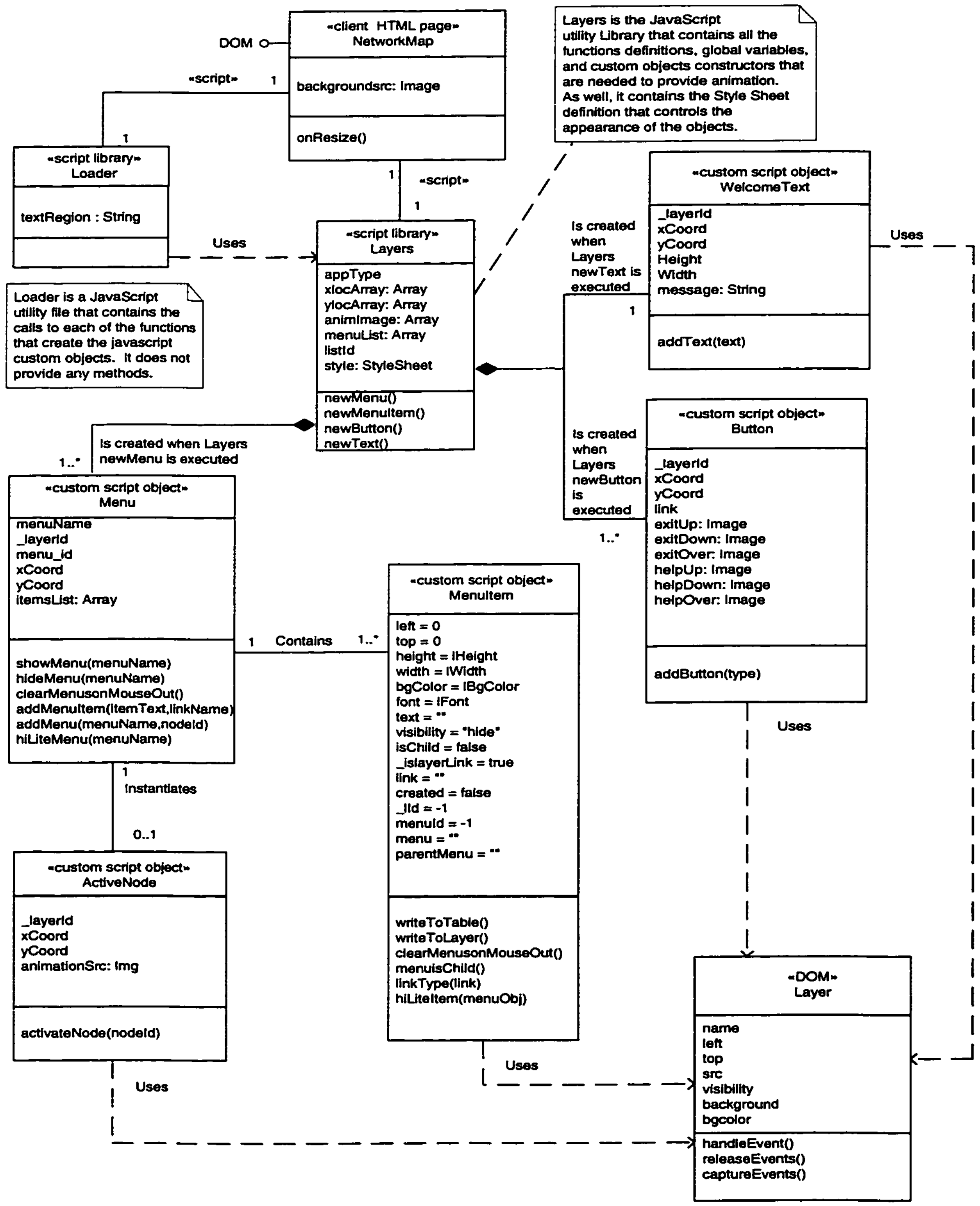


Figure 3.4 Class Diagram for Network Map HTML Page.

3.3 Roadmap Page Design

The Roadmap HTML page is a static, animated page that displays the content of the specific section of the documentation library corresponding to the product node, release, and platform the user selected in the previous Network Map page menus. The page essentially consists of different visual elements: a background image; Search, Index, Help, and Exit animated buttons; roadmap background image; roadmap columns; animated roadmap column books; book pop-up animated text boxes; etc. Figure 3.5 on page 48 shows a snapshot of this Roadmap page. Each of the visual elements have predefined locations on the screen, and in the case of the roadmap columns and books, they include text titles that vary for each of them. Hence, all of the different Roadmap pages have a standard look and behavior; what makes them different from each other is the text content they display on top of the roadmap images and of the book pop-up boxes, and the PDF document links implemented for each of the Books and Index buttons.

The generation of the Roadmap image is a complex process that depends on a lot of variables; a large amount of input information needs to be processed in order to render the final image as well as to provide the required animation. I decided to implement the drawing of the Roadmap animated image via a Java applet; Java offers much more object-oriented features to handle better that process than JavaScript. Because of backward compatibility issues, the Java Abstract Window Toolkit library (AWT) was used instead of the Java Foundation Classes (JFC) Swing. The GUI buttons as well as the rest of the GUI components were implemented from scratch, using the drawing capabilities of the AWT Graphics class. Although this is longer to develop, we get the greatest amount of flexibility and get to control every single aspect of the components' appearance and behavior.

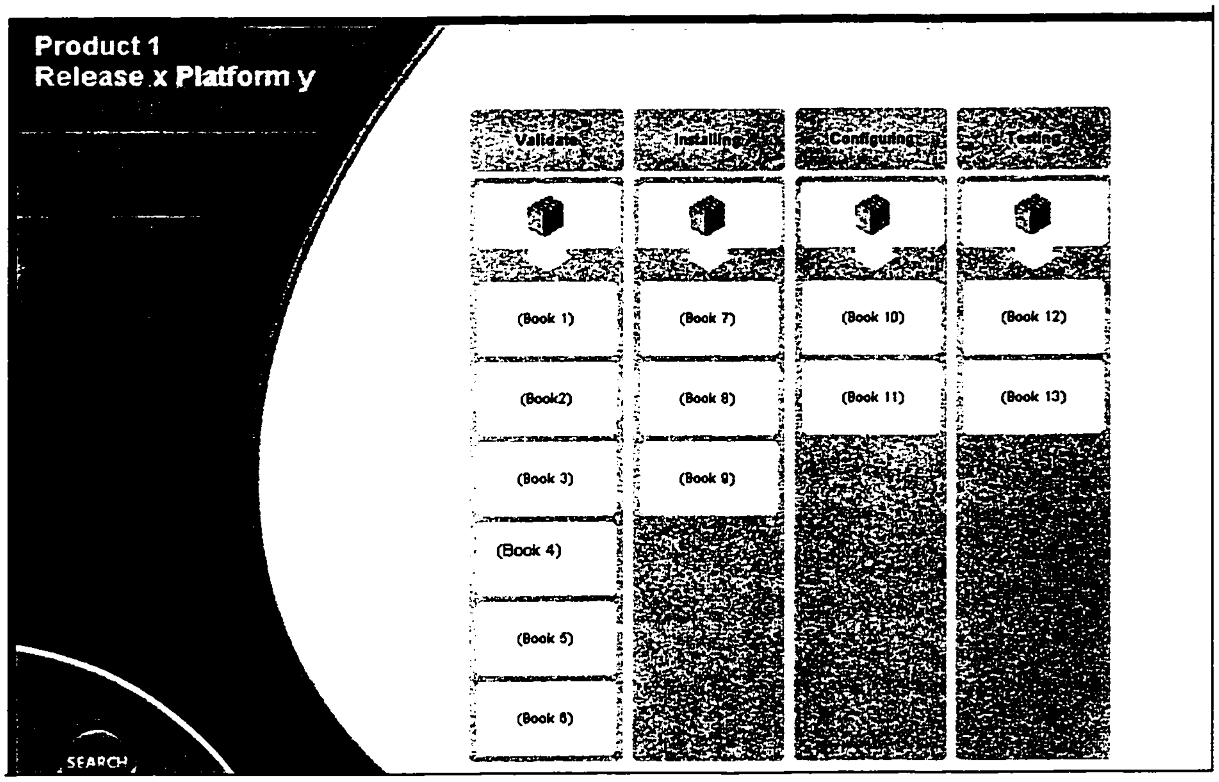


Figure 3.5 Roadmap Page.

3.3.1 Structural Model: Class Diagrams

The applet is composed of several classes that share different responsibilities. There is a main class, *RoadmapManager*, which controls the rendering of the Roadmap image into the screen and is also in charge of getting all the information from the Model, which is specified via parameters tags in the HTML page. All the events fired while the user interacts with the GUI, mostly mouse clicks and mouse movements, are controlled by the class *EventHandler*. *RoadmapManager* communicates the events coming from the system to the *EventHandler*, which handles all events and delegates work to the other classes.

The classes that make up the applet can be organized in three different groups, represented by the layers Presentation, Application, and Foundation. We can see how the different classes are classified into these groups in Figure 3.6 below.

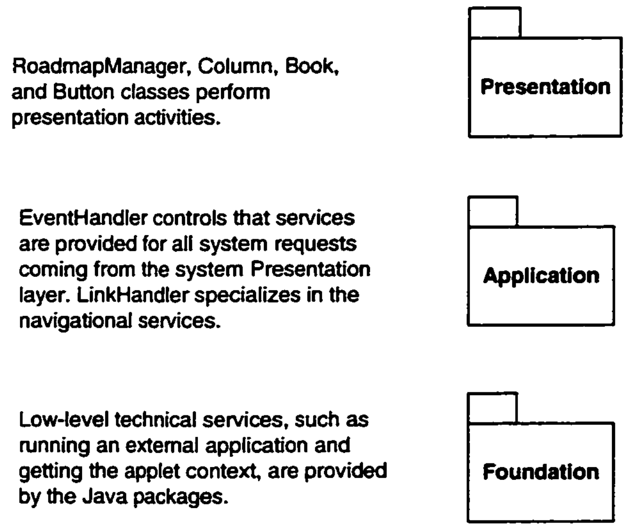

RoadmapManager, Column, Book, and Button classes perform presentation activities.
Presentation
EventHandler controls that services are provided for all system requests coming from the system Presentation layer. LinkHandler specializes in the navigational services.
Application
Low-level technical services, such as running an external application and getting the applet context, are provided by the Java packages.
Foundation

Figure 3.6 Roadmap Applet Structure.

Figure 3.7 on next page shows the class diagram for the Roadmap page applet. The *RoadmapManager* class extends the native Java *Applet* class and is responsible for controlling the rendering of the Roadmap image; it controls the creation of the other objects and communicates the input information needed to create and display all the visual elements that make up the final Roadmap image. This class has a ***creator*** and ***uses*** relationship with the *EventHandler* class, which is responsible for handling the system

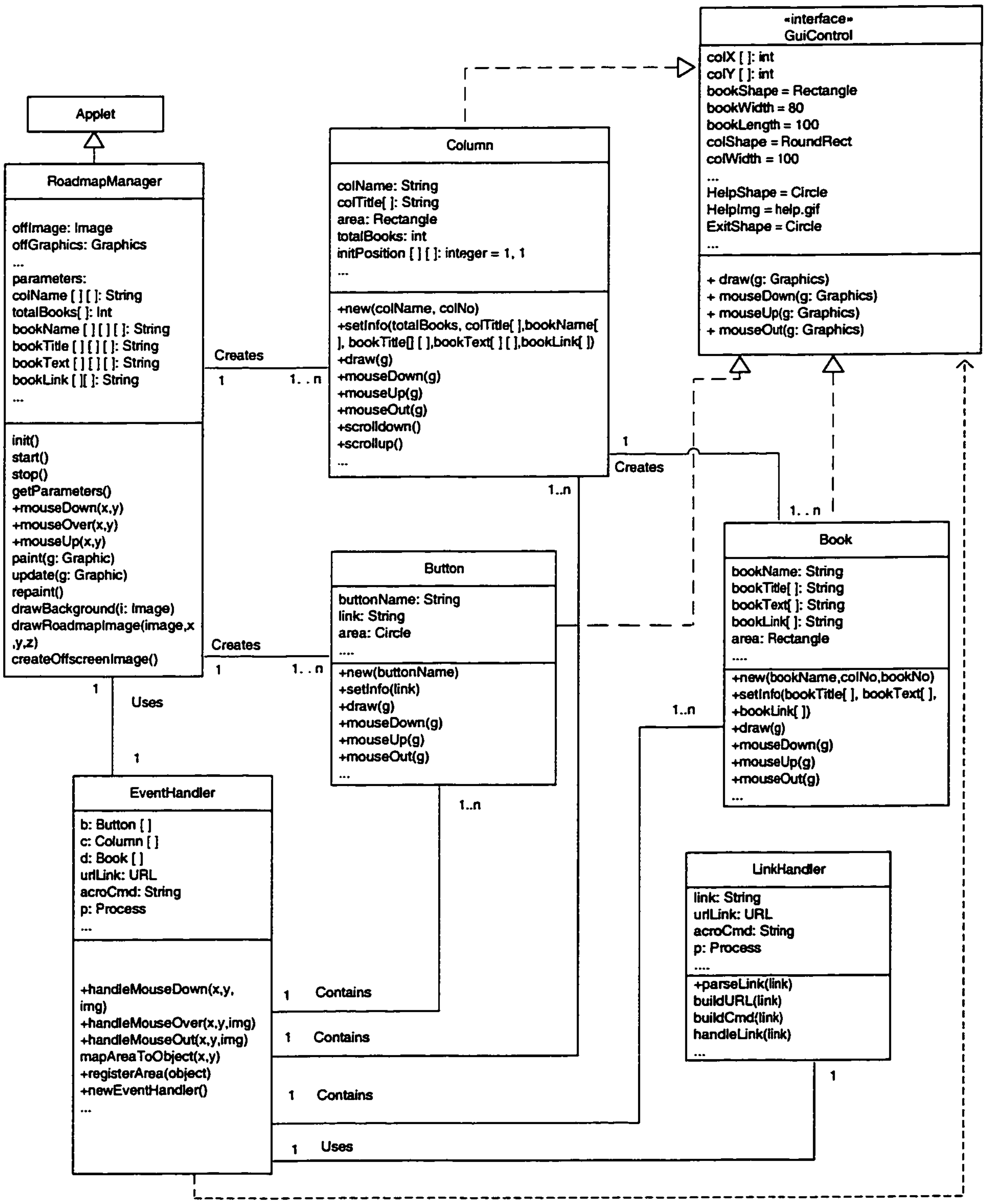


Figure 3.7 Class Diagram for Roadmap Applet.

events, as well as with the key presentation classes, *Column* and *Button*. Since the *Column* object contains the *Book* objects, a *creator* relationship is established between them. The *creator* relationship--although not shown on the diagram--is also established from the *EventHandler* towards the *LinkHandler*, since the latest helps to the handling of certain system events.

The classes that represent a visual element of the GUI, like the *Column* and *Button* classes, do not talk between themselves; they communicate only with the controller or event handler. The *Book* object talks to the *Column* object since it is contained by this one. All these classes offer services specialized on the drawing of each different visual component and on handling its animation.

EventHandler keeps a reference to each visual component created and also stores their sensitive area coordinates which are needed in order to delegate a given system event. Since *EventHandler* is playing the role of Information Expert--shown on the class diagram as a *contains* relationship--it is assigned with the task of handling all events. To prevent having a bloated controller, this class delegates the presentation tasks to the visual component classes and the navigational tasks to the *LinkHandler* class. A *uses* relationship between the *EventHandler* and the *LinkHandler* is therefore established.

A very important structural component is the *GUIControl* interface that outlines the common behavior, draw and animation methods shared by all of the visual components. Each method's signature is the same for all the classes however the implementation details are left to them to handle. Consequently, a realization relationship from the *Book*, *Column*, and *Button* objects, to the *GUIControl* interface is illustrated on the class diagram. As well, a dependency relationship is shown from the *EventHandler* to the *GUIControl* interface, since the *EventHandler* expects all of the presentation classes to implement the methods specified by that interface. Finally, the *GUIControl* interface holds constant values for the rendering of the roadmap image, and makes them globally available to the classes that implement it. It constitutes a partial business model for this subsystem.

The distribution of responsibilities for each of the classes was inspired again by the MVC design pattern; however, it was not strictly applied. The *RoadmapManager*, which represents the applet class itself, is handling View related requests as well as executing some Controller tasks. It receives the system events and communicates them to the *EventHandler*--the other controller-- which handles them. The View is also constituted by all the GUI visual components *Book*, *Button*, and *Column*. The Model, which "is responsible for knowing and maintaining the state of the component" [Weber 500], is represented in part by the *param* tags in the HTML file that acts as a batch file containing the runtime initialization parameters; the other part of the Model is represented by the *RoadmapManager* which knows which system event has been fired. Moreover, the Model also has constant values that are stored in the *GUIControl* interface and are available to all the classes that implement it.

High Level Class Diagram

Figure 3.8 on next page shows a high level class diagram for the Roadmap applet. The classes expressly designed for this Roadmap applet application, are shown in its expanded form, though the attributes and operations are omitted; the classes provided by the Java packages are shown as rectangular icons. The dependency relationships between the Roadmap applet's specialized classes and the classes provided by the different Java packages are shown here to emphasize the effects a change to the Java utilities can have on the system.

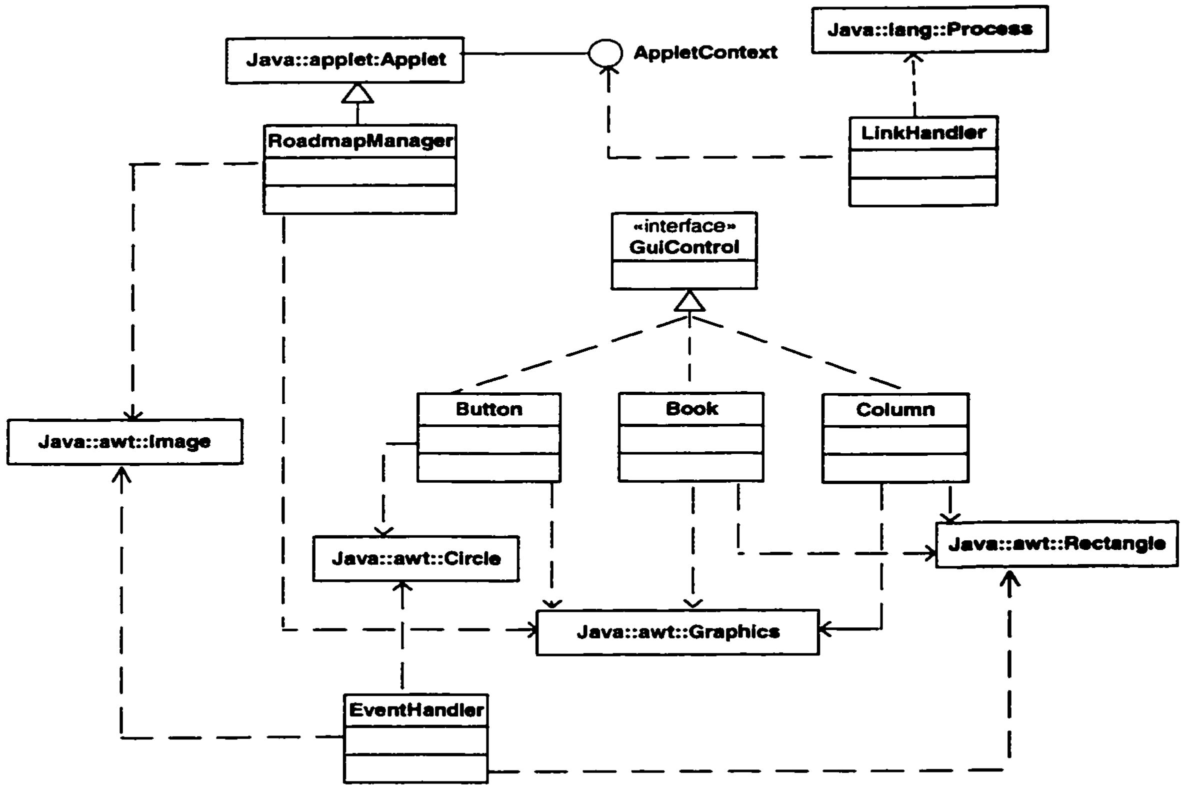


Figure 3.8 High Level Class Diagram for Roadmap Applet.

The *Graphics* and *Image* classes provided by the Java Abstract Window Toolkit package (AWT) are extensively used by most of the classes. The abstract class *Graphics* provides the graphic context that allows the applet to draw onto off-screen images. The abstract class *Image* is the superclass of all classes that represent graphical images; it is being used to build the final image of the Roadmap and to import gif files. Other two key AWT utility classes used are *Rectangle* and *Circle*, which represent an area in a coordinate space. The *AppletContext* interface provided by the *Applet* class is being implemented here to link to other HTML pages. Since the applet is being housed by a browser, it is detected as being the applet context and receives the instruction to load a URL. The *Process* class provided by the Java *lang* package provides the mechanism to run the external Acrobat Reader application. The *exec* method returns an instance of a subclass of *Process*; it is used to obtain further information about the new process, e.g., whether the call succeeded or an error was returned by the Acrobat Reader application after it was open, etc.

3.3.2 Behavioral Model: State and Sequence Diagrams

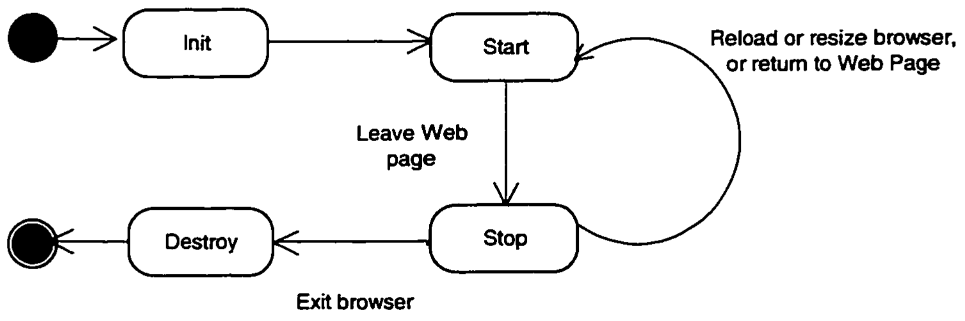


Figure 3.9 Applet Lifecycle.

Figure 3.9 above represents the states--or methods--an applet goes through during its existence [Weber 240]. Besides these basic methods, there are other two important methods: *paint* and *update*. *Paint* is called when the contents of the component should be painted in response to the component first being shown or damage needing repair. The AWT calls the *update* method in response to a call to *repaint* or *paint*. By default, it clears the panel and then calls *paint* which causes a flickering. That is why this *update* method is overridden to simply call *paint*.

Figure 3.11 on page 56 illustrates the *init* method; the *registerArea* message is used by each view object to register its existence with the *EventHandler* class, so that when a mouse event is triggered by the user, the *EventHandler* knows who to notify for an update of the screen. Similarly, Figure 3.10 below shows how each view object is handling the work involved in the drawing of itself and of its attached components.

1. The applet RoadmapManager object requests the drawing of the background image from the offscren Graphic object.

2. The manager then calls the method draw( ) from the Column object, which will update the offscreen Graphic object being passed as a parameter.

3. Each column calls the draw( ) method for each book in its collection. The drawing of each book completes the drawing of the column itself.

4. Once control comebacks to the manager, it calls the draw( ) method for each Button on the collection.

5. The manager renders the applet's image. It internally calls the drawImage() method of the corresponding offscreen Image object.

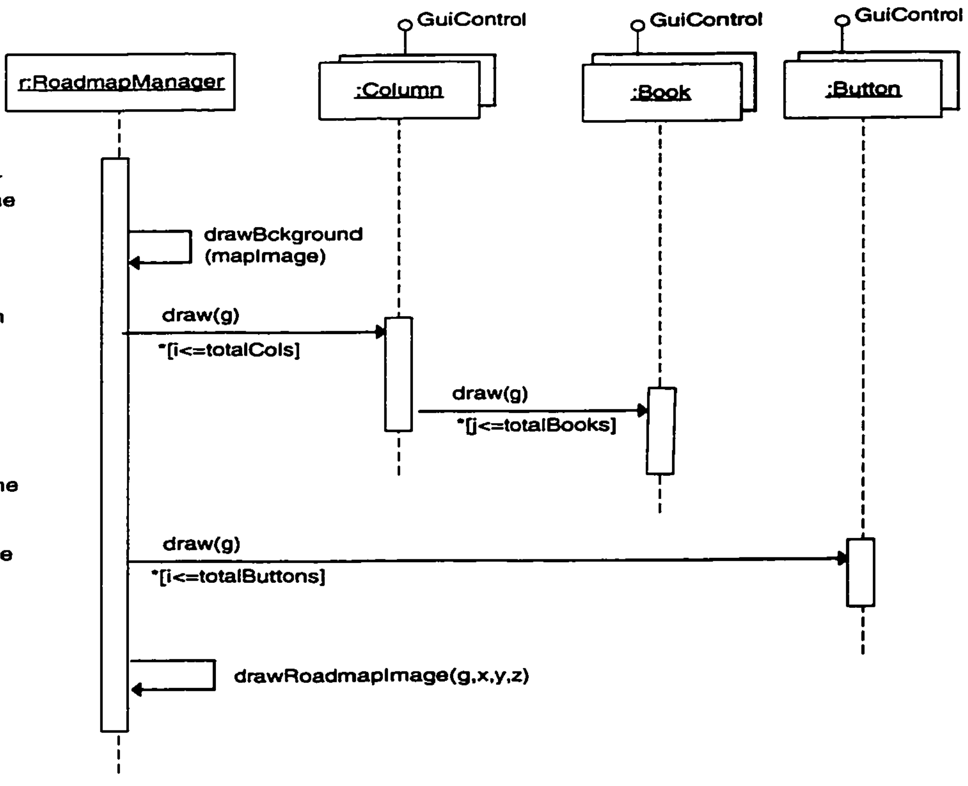
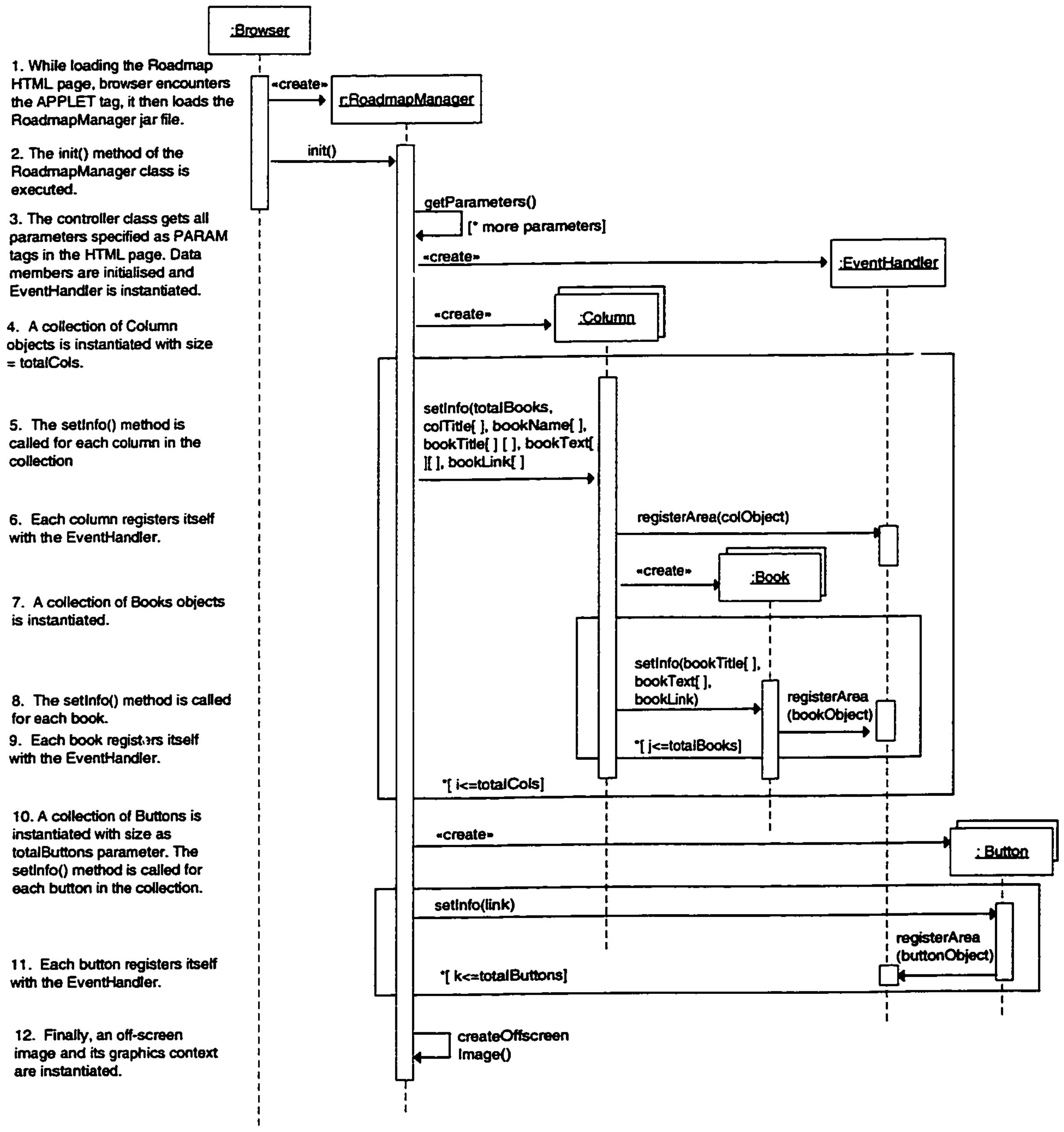


Figure 3.10 Sequence Diagram for *paint* method of the Roadmap Applet Class.



1. While loading the Roadmap HTML page, browser encounters the APPLET tag, it then loads the RoadmapManager jar file.

2. The init() method of the RoadmapManager class is executed.

3. The controller class gets all parameters specified as PARAM tags in the HTML page. Data members are initialised and EventHandler is instantiated.

4. A collection of Column objects is instantiated with size = totalCols.

5. The setInfo() method is called for each column in the collection

6. Each column registers itself with the EventHandler.

7. A collection of Books objects is instantiated.

8. The setInfo() method is called for each book.

9. Each book registers itself with the EventHandler.

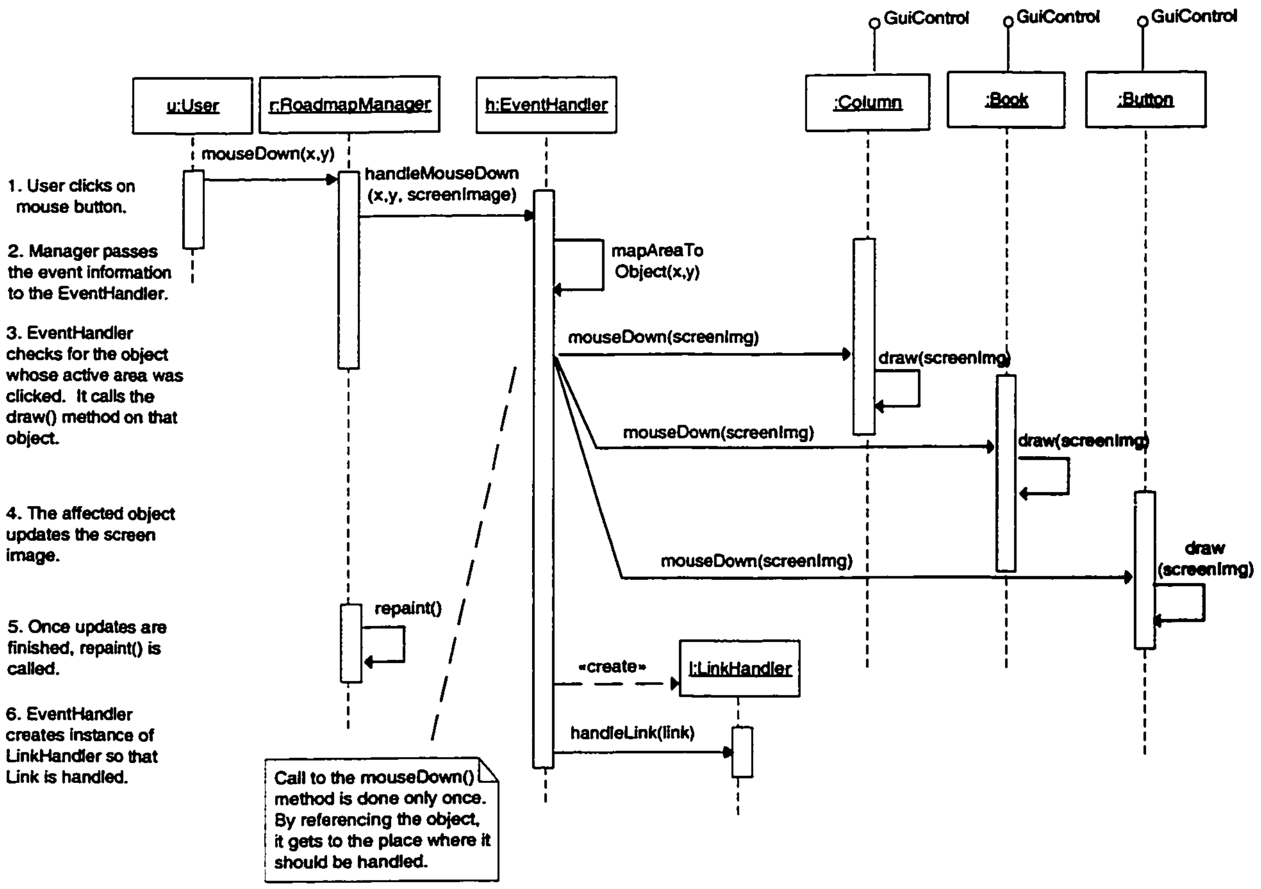
10. A collection of Buttons is instantiated with size as totalButtons parameter. The setInfo() method is called for each button in the collection.

11. Each button registers itself with the EventHandler.

12. Finally, an off-screen image and its graphics context are instantiated.

Figure 3.11 Sequence Diagram for *init* Method of the Roadmap Applet Class.

Figure 3.12 below, illustrates the sequence of events that get triggered when the user clicks on a mouse button. It is clear that responsibility for handling the event resides on the *EventHandler* object. *EventHandler* forwards the event to the corresponding view object so that it updates by itself the roadmap image on the screen. Once that has been done, and for this particular event of *mouseDown*, the *handleLink* method of the *LinkHandler* object is called, with a Link parameter the *EventHandler* got from the corresponding view object.



GuiControl
GuiControl
GuiControl
u:User
r:RoadmapManager
h:EventHandler
:Column
:Book
:Button
mouseDown(x,y)
handleMouseDown (x,y, screenImage)
mapAreaTo Object(x,y)
mouseDown(screenImg)
draw(screenImg)
mouseDown(screenImg)
draw(screenImg)
mouseDown(screenImg)
draw (screenImg)
repaint()
«create»
l:LinkHandler
handleLink(link)
1. User clicks on mouse button.
2. Manager passes the event information to the EventHandler.
3. EventHandler checks for the object whose active area was clicked. It calls the draw() method on that object.
4. The affected object updates the screen image.
5. Once updates are finished, repaint() is called.
6. EventHandler creates instance of LinkHandler so that Link is handled.
Call to the mouseDown() method is done only once. By referencing the object, it gets to the place where it should be handled.

Figure 3.12 Sequence Diagram for *mouseDown* event on CD-ROM GUI Interface.

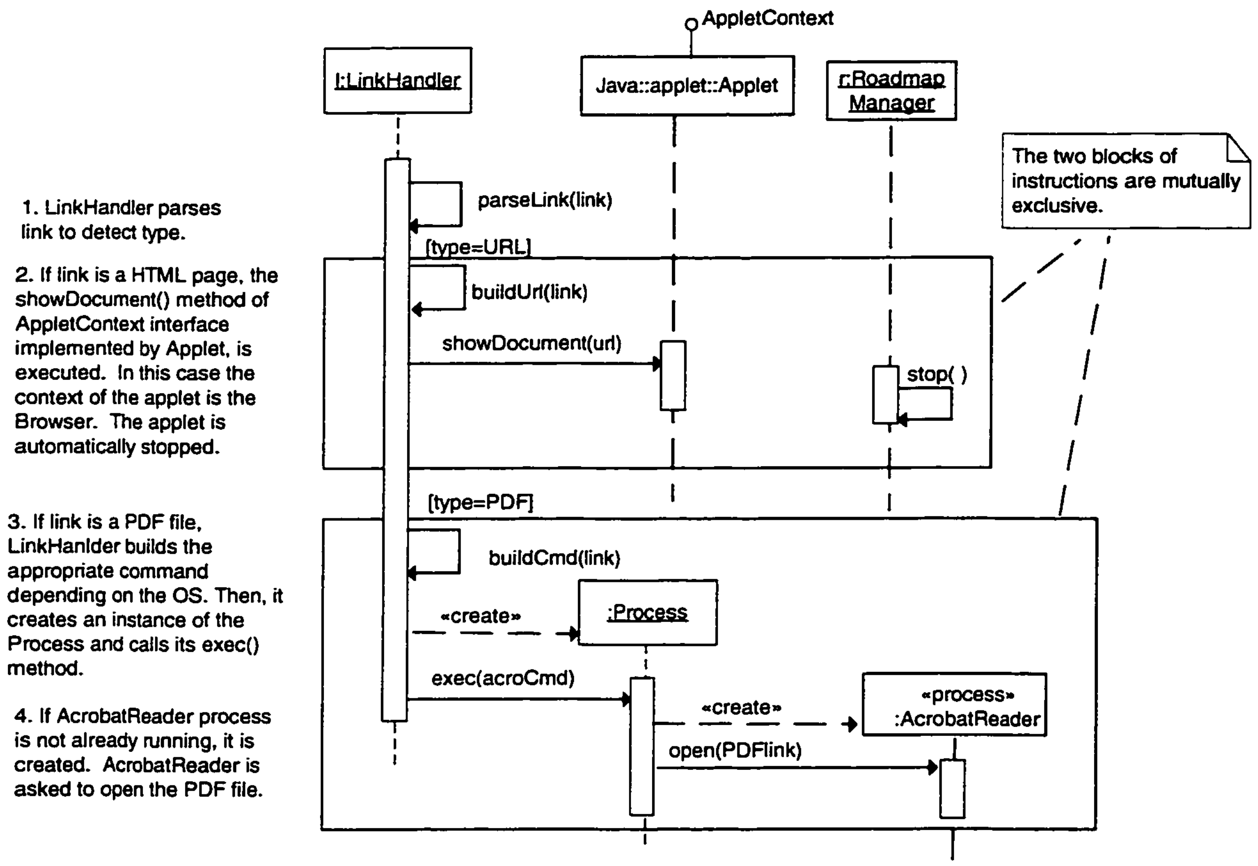

AppletContext
l:LinkHandler
Java::applet::Applet
r:Roadmap Manager
The two blocks of instructions are mutually exclusive.
1. LinkHandler parses link to detect type.
parseLink(link)
[type=URL]
2. If link is a HTML page, the showDocument() method of AppletContext interface implemented by Applet, is executed. In this case the context of the applet is the Browser. The applet is automatically stopped.
buildUrl(link)
showDocument(url)
stop( )
[type=PDF]
3. If link is a PDF file, LinkHanlder builds the appropriate command depending on the OS. Then, it creates an instance of the Process and calls its exec() method.
buildCmd(link)
«create»
:Process
exec(acroCmd)
4. If AcrobatReader process is not already running, it is created. AcrobatReader is asked to open the PDF file.
«create»
«process» :AcrobatReader
open(PDFlink)

Figure 3.13 Sequence Diagram for *handleLink* Method of *LinkHandler* Classs.

Figure 3.13 above, illustrates the *handleLink* method of the *LinkHandler* class. *LinkHandler* first detects what kind of link the *Book* or *Button* object is linking to. It then translates it into the proper type: a URL or an OS command to run Acrobat Reader. If the link was a URL, it calls the method *showDocument* available from the *appletContext* interface. Since the applet context is an HTML page in a browser, the browser shows the new HTML document and the *stop* method of the *RoadmapManager* applet class is automatically called. If link was to a PDF file, *EventHandler* creates an instance of the

Java *Process* class and executes its *exec* method with the OS command as a parameter. The Java *Process* class creates a separate process to run the Acrobat Reader application.

3.3.3 Security Model

Although not shown on Figure 3.13 illustrating the *handleLink* method on previous page, the Java Security Model will be used when the applet tries to launch the Acrobat Reader application. This happens because applets, even when not loaded over the net, are not allowed to access the local machine's files and disks. Java-enabled browsers use the applet class loader to load applets specified with *file:URLs*; so these applets are subject to the restrictions and protections of the applet security manager. Therefore, the applet will need to be signed with a signing tool, either from Netscape or from Sun; Microsoft IE signing tools do not support the Java Security model.

Once the applet is signed, the first time the user runs the application and navigates to the Roadmap page, he will be asked to grant the Security Certificate. The user must select 'Grant Always' so that he will not be prompted anymore.

3.4 Architecture: Modeling the Executable Release

All the JavaScript functions that make up the animation of the Network Map home page are being physically packaged in two files: the *Layers.js* and the *Loader.js*. Concerning the Roadmap HTML pages, all of them need access to the Roadmap Applet main and related classes. Because of the advantages of zipping files, all of the classes that make up the applet will be packaged in a JAR file format. This will bring the advantage of fast access, reduced size, security, and portability. Portability is a great advantage, since the JAR file format is browser independent, as is Security, since JAR files allow the user to verify the origin of the applet and mark it as trusted. Using the Java security model, this verification is done in a single browser-independent way.

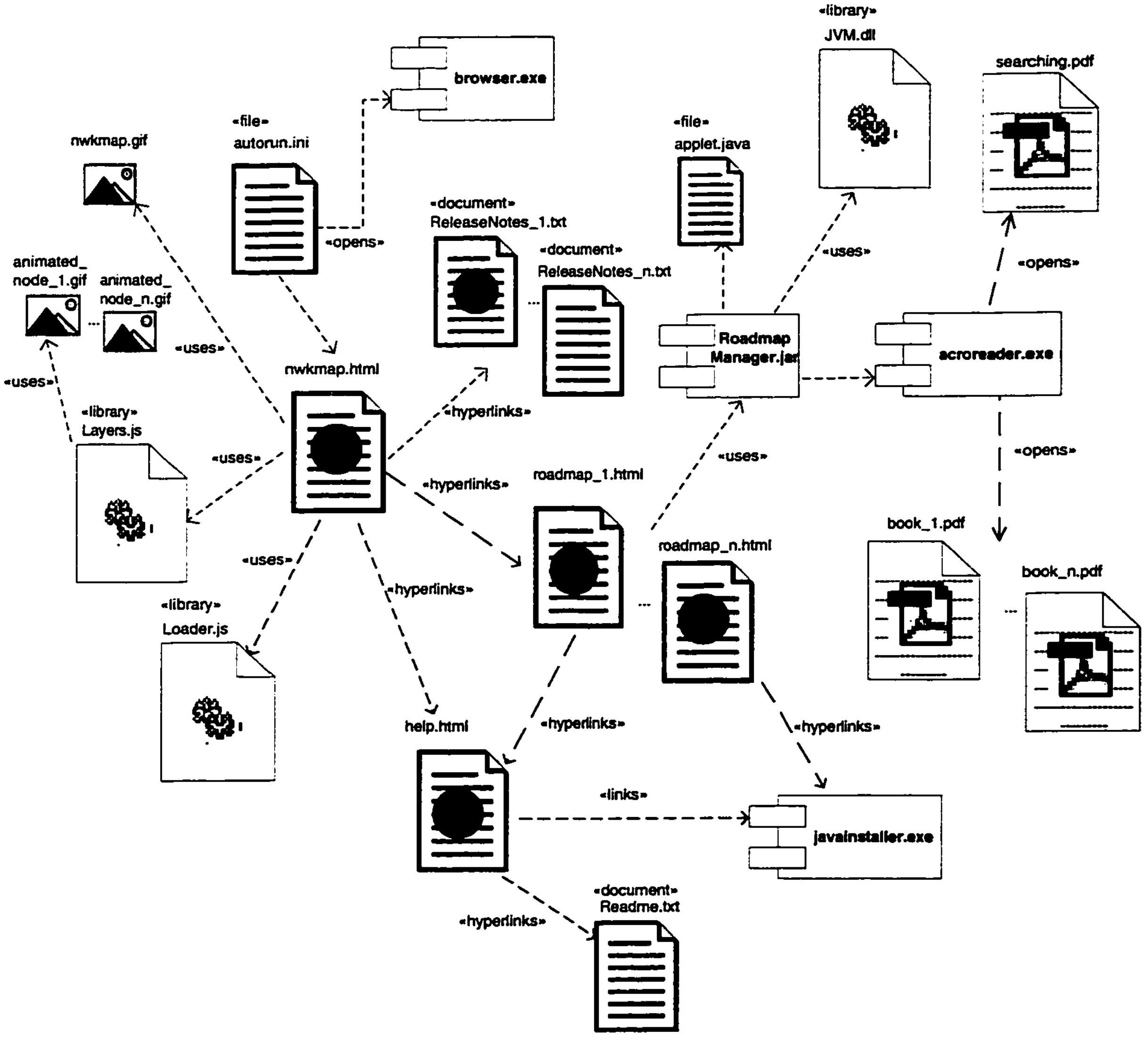


Figure 3.14 Component Diagram for CD-ROM GUI.

Figure 3.14 on previous page shows the configuration of the CD-ROM GUI system. There are three executable components:

- *Browser.exe* identifies the user's default browser application used to render HTML pages.
- *Acroreader.exe* represents the Acrobat Reader application used to open the PDF files.
- *Javainstaller.exe* represents the Java Install Shield used to install the Java virtual machine.

The *autorun.ini* file is an initialization file that on the Windows OS platform gets automatically executed when the user inserts the CD-ROM into its corresponding drive. In the CD-ROM GUI system, it will launch the browser to start the Network Map home page, saved under the name *nwkmap.html*. This home page has a background gif showing the network of interrelated product nodes; it loads two JavaScript libraries, *Layers.js* and *Loader.js*. *Loader.js* calls the animation functions defined on *Layers.js*, with the appropriate parameters; these functions create different layers like the welcome introductory text region, the releases and platforms menus, and the Help and Exit buttons. Another function, used to animate the active product nodes, is represented by the dependency relationship between *Layers.js* and product nodes gifs. The Help button links to the *help.html* file. The releases or platforms menu options link to its corresponding *roadmap.html* file.

The *roadmap.html* page file contains a reference to the Roadmap applet that draws and controls the content of the page. Using special Java plug-in tags, *OBJECT* and *EMBED*, the browser is forced to launch the Sun's Java virtual machine to run the applet; if it is not installed in the user's machine the user is directed to install it from a Web site--Sun's by default. The applet consists of a set of Java compiled classes, packaged into a JAR format; the corresponding source code is represented by the *applet.java* file component. A dependency relationship between the *RoadmapManager.jar* component and the Acrobat Reader executable component represents the calls the applet makes to the reader, launching it with a PDF file as parameter. This occurs whenever the user clicks on one of the roadmap book buttons, on the Search, or on the Index buttons.

The *help.html* file hyperlinks to the Readme file that contains a User Guide to install manually the software required: a Microsoft or Netscape browser, and the JRE. It also provides a link to the Java installer included in the CD-ROM.

CHAPTER 4 CD-ROM PUBLISHING SYSTEM DESIGN

4.1 Design Rationale

The CD-ROM Publishing system (CD-ROM PUB) is the other component of the EDSTDL system. It is essentially a subsystem that provides the services required to automate the production of the CD-ROM GUI subsystem. The user's interaction with the system is limited to making menu selections, entering some input parameters, launching the process, and waiting for the results. Following are the system's principal constraints:

- The user expects to use the publishing system in a PC environment.
- The system needs to be developed quite quickly and with a minimum budget.
- The visual interface component of the CD-ROM GUI might need to be changed in the future, as new customer's needs appear.

I have developed an object-oriented design focused in creating objects that can be easily changed and reused in other applications. To facilitate the future maintenance of the system was one of the forces that strongly affected the distribution of responsibilities among the classes. Even though the design was done having in mind a Visual Basic (VB) implementation, it can still be implemented in other object-oriented languages, like C++ or Java. It is important to mention that recently released Microsoft's VB.NET is now fully object-oriented with the inclusion of class inheritance. Although with this new release the Component Object Model (COM) technology created by Microsoft has been abandoned in favor of the .NET platform, VB still fully supports the interaction with ActiveX components like those included in the Microsoft Office applications. COM-based OLE Automation enabled our application to control the OLE-enabled application Microsoft Excel.

The design is centered in the Domain layer components which are presented at a very low level of detail. The Presentation layer is briefly presented since it is practically done by making use of the predefined classes that VB offers: forms, buttons, windows, and other kind of widgets. Moreover, the Domain layer is where most of the operations are happening; once the user selects the service he needs, all the processes to fulfill his request are done behind the scenes, like in a batch system. He only receives a message back from the application once everything finished and a log of events is generated.

4.2 Modeling the Structure of the System

The fundamental structure of the CD-ROM PUB interactive application is based on the architectural pattern Model-View-Controller (MVC). This pattern divides an interactive application into three components. "The model contains the core functionality and data. Views display information to the user. Controllers handle user input. Views and controllers together comprise the user interface." [Buschmann 125].

4.2.1 Presentation Layer Structure

Figure 4.1 below presents a high level class diagram of the Presentation layer. It illustrates how the GUI controls provided by Visual Basic are customized to create the visual interface of the system. The *Form*, *Menu*, *Button*, and *DirectoryList* VB classes are presented as parent classes of the actual classes used on the windowed interface of the CD-ROM PUB system.

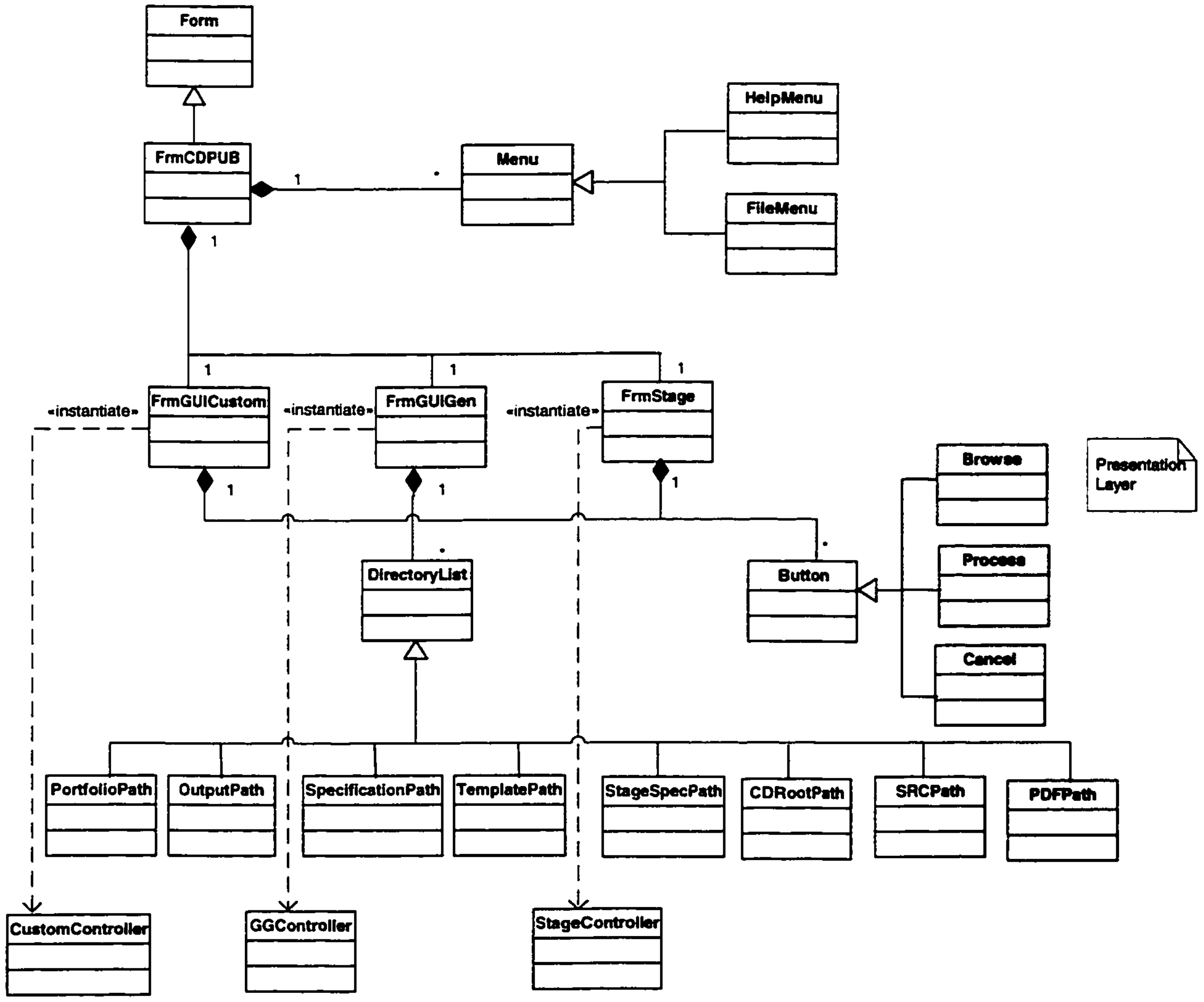


Figure 4.1 Class Diagram for the Presentation Layer of the CD-ROM PUB System.

The CD-ROM PUB system has a multi-document form (MDI) *FrmCDPUB* as start-up object. The MDI form presents a menu choice to the user; depending on the menu option that is selected--Customize GUI, Generate GUI, or Create Stage Area--the corresponding child form will be loaded. These child forms consist of a window with buttons to browse a folder, to launch the process, or to cancel it. As well, they present different directory list controls that are used to indicate where the input files are placed and where the output files should be stored.

During its initialization process, each of the child forms creates an instance of a controller class: *CustomController*, *GGController*, or *StageController*.

4.2.2 Domain Layer Structure

Figure 4.2 on page 67 presents the class diagram for the Domain layer of the CD-ROM PUB system. Some of the classes that appear in the Object Design Model are named after their counterparts from the Domain Model. However, most of them were created expressly to fulfill the needs of an object-oriented design, by applying the Pure Fabrication design pattern. It is interesting to note that some of the decisions done during the requirements-definition phase regarding class responsibilities did not longer apply within the system design context. The final distribution of responsibilities among the different classes was influenced by the following enabling techniques for software architecture [Buschman]:

- Abstraction
- Encapsulation
- Information Hiding

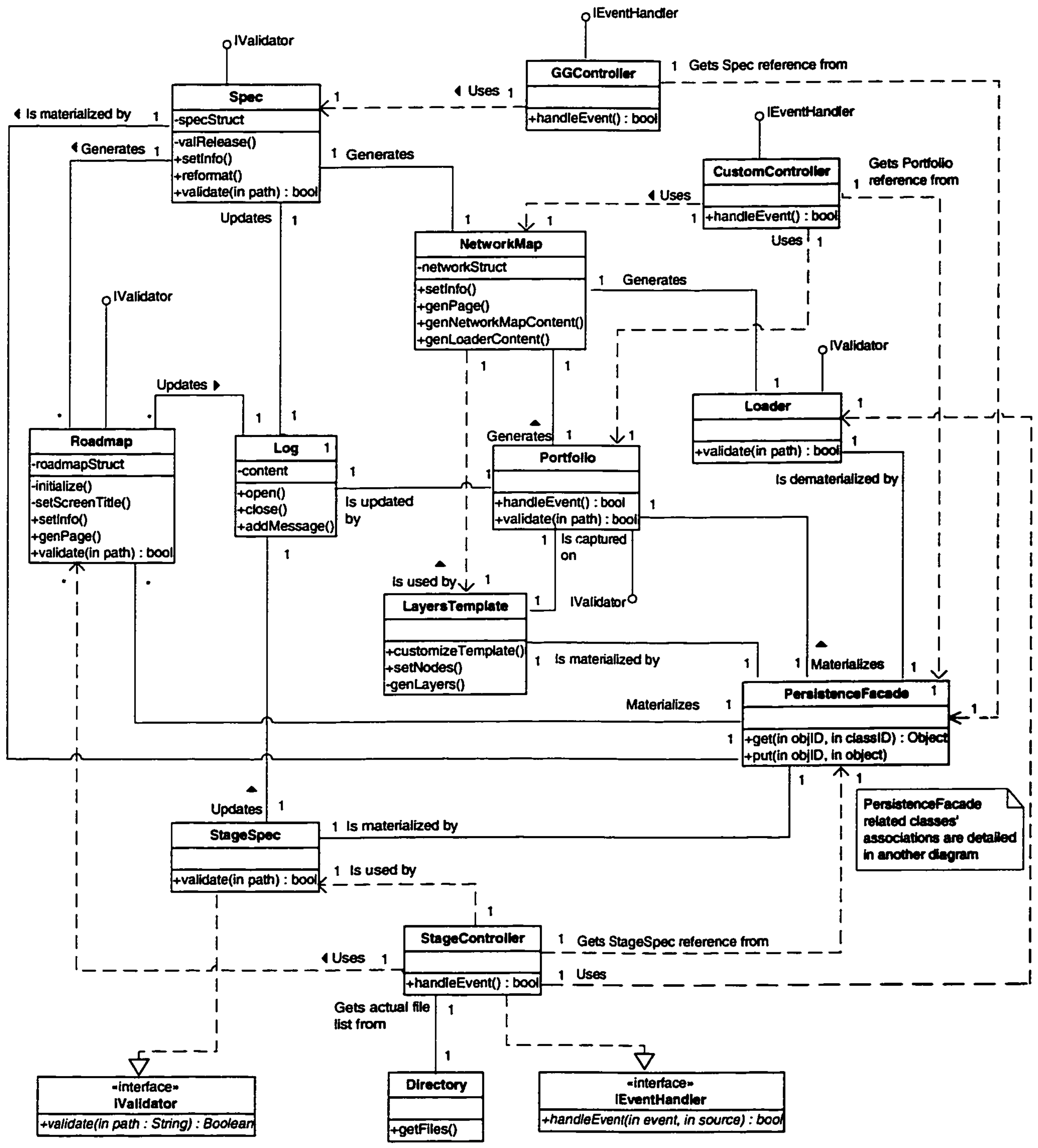


Figure 4.2 CD-ROM PUB System Domain Layer Class Diagram.

- Modularization

- Separation of Concerns

- Coupling and Cohesion

- Separation of Interface and Implementation

- Divide and Conquer

The design of the Domain Layer involved the use of several GRASP design patterns. GRASP is an acronym that stands for General Responsibility Assignment Software Patterns. "They describe fundamental principles of object design and responsibility assignment, expressed as patterns." [Larman 220].

The GRASP Controller pattern was applied in the creation of the event handler classes *CustomController*, *GGController* and *StageController*. These controllers represent as well the event handler component of the MVC design pattern and help to separate the presentation from the application layer. In order to promote High Cohesion--another GRASP design pattern--as well as to avoid bloating, a controller was created for each of the use case scenarios. All controllers implement the *EventHandler* interface that specifies the *handleEvent* operation signature with the source form as an argument and returning a general validation status. Although not specified in the signature, this operation must generate a log file.

Another GRASP pattern, the Information Expert, was applied when assigning responsibilities to the controller's helper classes: *Spec*, *Portfolio*, *Roadmap*, *Loader*, and *StageSpec*. In the class diagram on previous Figure 4.2, the using relationships between each controller and its helper classes are presented as dependencies. Since these classes have all the information required to validate the content of the different GUI specifications;

they were assigned with the responsibility of validating the information, from a business rules perspective. All of the controller's helper classes implement in their own way the interface *IValidator*.

The CD-ROM PUB subsystem makes extensive use of I/O based operations. Most of the objects representing the output CD-ROM GUI content like the Network Map and Roadmap HTML pages, Layers and Loader JavaScript files, need to be stored in the user PC's hard disk. As well, the CD-ROM GUI input specification files need to be retrieved from the hard disk and materialized into the different objects *Spec*, *Portfolio*, *LayersTemplate*, or *StageSpec*. The tasks of materializing and de-materializing these objects are fulfilled by several classes implementing the interface *IMapper*. *IMapper* defines two operations, *get* and *put*, that retrieve and store an object from and into the hard disk. A different mapper class was needed for each persistent object and each of them implements the *get* and *put* methods in their own way. The Low Coupling GRASP pattern and the Façade pattern were applied by creating the *PersistentFacade* class. This class provides a unified interface for the different mapper classes and the classes underneath them. All the objects refer only to one class instead of accessing directly each of the different mapper classes, achieving with this low coupling. Although we are not presenting the design of a highly sophisticated persistence system, the classes built here constitute the base of a simple disk storage subsystem that could be easily reused and extended in other applications. A dependency relationship between the controllers and the *PersistenceFacade* class is emphasized on previous Figure 4.2.

Certain classes, like *PersistenceFacade* and *Log*, need to support global visibility as well as a single access point to a single instance. Instead of passing around an instance as a parameter or initializing the objects that need visibility to it with a permanent reference, the Singleton design pattern is applied. This is shown on the class diagram of previous Figure 4.2 on page 67, by specifying a multiplicity of one in the *Log* and *PersistenceFacade* class icons.

The Persistent Object Storage Class Diagram on Figure 4.3 below, presents the different classes that lie under the *PersistenceFacade* class. This façade class was created by applying the GRASP pattern of Pure Fabrication. It does not correspond to anything in the Domain Model; it is there for the convenience of the software developer.

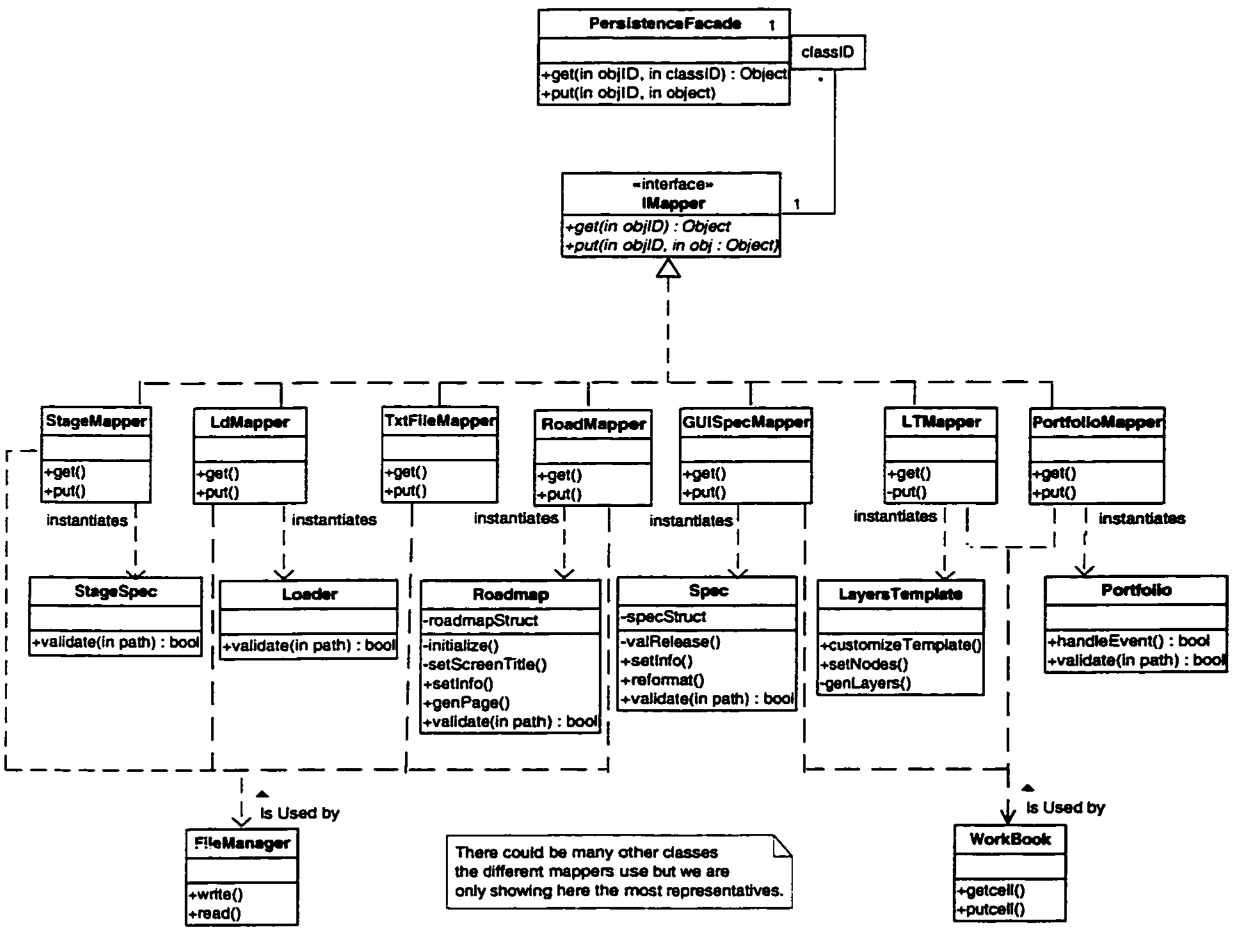


Figure 4.3 Persistent Object Storage Class Diagram.

4.3 Modeling the Behavior of the System

In the following pages we will see how all the classes previously presented interact with each other in the realization of the major use cases. Sequence diagrams are used to illustrate the passing of messages over time. The messages exchanged between the Presentation layer classes are only detailed in the first sequence diagram, the GUI Customization use case. The same kind of interaction applies for the rest of the publishing options.

4.3.1 GUI Customization Publishing Option

Earlier in chapter 2, the GUI Customization use case was presented as an essential service the CD-ROM PUB system must provide. Once the need for a new portfolio of products arises, the technical illustrator creates the artwork required for the Network Map HTML page of the CD-ROM GUI. He then uses the GUI Customization publishing option to automatically customize the CD-ROM GUI.

The CD-ROM GUI subsystem design presented in chapter 3 specifies a JavaScript Layers file containing the library of JavaScript functions that animate the Network Map home page. The GUI Customization publishing option automatically makes the necessary adjustments to the Layers file in order to reflect the new portfolio of products. The product node's position might change when introducing a new product in the Network Map. As well, the background image might change in the case of a completely new portfolio.

Figure 4.4 on page 72 presents the interaction diagram for the GUI customization option. The actor of this use case, the technical illustrator, selects the GUI Customization

1. Illustrator selects CustomizeGui menu option.

2. After system shows the GUI Customization form, user browses to locate portfolio artwork xls file and output folder. He then clicks button to launch customization process.

3. The event is sent to the controller. Controller gets an instance of the PersistenceFacade object and calls its get() method with the portfolio class as a parameter.

4. PortfolioMapper creates an instance of an Excel ActiveX object and reads the contents of the portfolio artwork file. It then instantiates the Portfolio object.

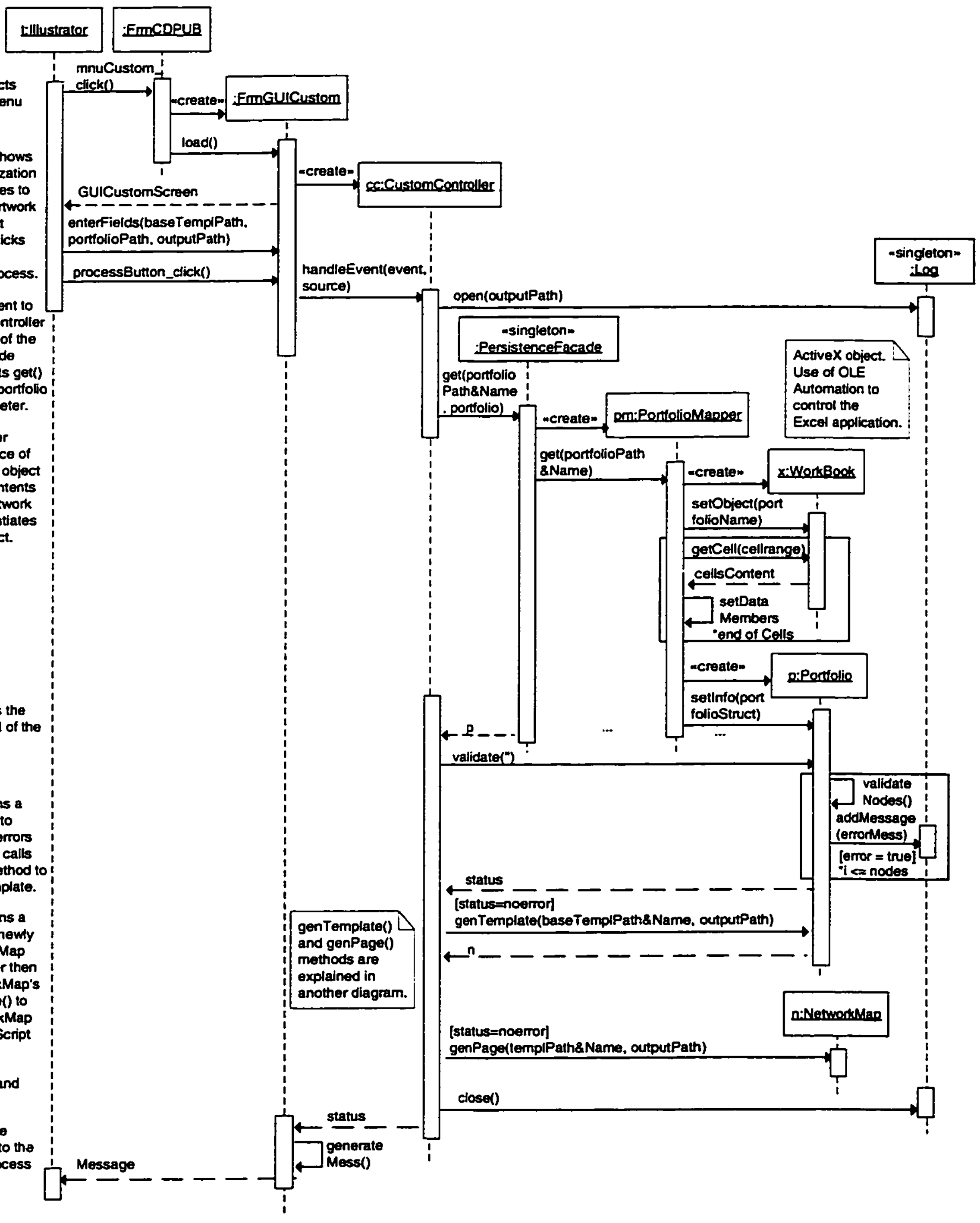
5. Controller calls the validate() method of the Portfolio object.

6. Portfolio returns a validation status to Controller. If no errors found, Controller calls the Portfolio's method to generate the template.

7. Portfolio returns a reference to the newly created NetworkMap object. Controller then calls the NetworkMap's method genPage() to have the NetworkMap HTML and JavaScript files generated.

8. Log is closed and saved.

9. The appropriate message is sent to the user once the process is finished.



t:Illustrator
:FrmCDPUB
mnuCustom_click()
«create»
:FrmGUICustom
load()
«create»
cc:CustomController
GUICustomScreen
enterFields(baseTemplPath, portfolioPath, outputPath)
processButton_click()
handleEvent(event, source)
«singleton» :Log
open(outputPath)
«singleton» :PersistenceFacade
ActiveX object. Use of OLE Automation to control the Excel application.
get(portfolio Path&Name , portfolio)
«create»
pm:PortfolioMapper
get(portfolioPath &Name)
«create»
x:WorkBook
setObject(port folioName)
getCell(cellrange)
cellsContent
setData Members
*end of Cells
«create»
p:Portfolio
setInfo(port folioStruct)
p
...
validate(*)
validate Nodes()
addMessage (errorMess)
[error = true]
*i <= nodes
status
[status=noerror] genTemplate(baseTemplPath&Name, outputPath)
n
genTemplate() and genPage() methods are explained in another diagram.
n:NetworkMap
[status=noerror] genPage(templPath&Name, outputPath)
close()
status
generate Mess()
Message

Figure 4.4 Sequence Diagram for GUI Customization Use Case, Part I.

menu option found on the MDI start up form. Once the *FrmGUICustom* form object is loaded, its corresponding event handler class, *CustomController*, is created. After the user enters all the required input parameters and presses the process button, *FrmGUICustom* object simply sends the *handleEvent* message to the controller.

CustomController object gets an instance of the *PersistenceFacade* singleton class and calls its *get* method with portfolio class as a parameter, to have the *Portfolio* object materialized. The *Portfolio* object captures the artwork specification previously captured in a spreadsheet by the illustrator. The *PersistenceFacade* class will in turn create an instance of the appropriate mapper object, in this case the *PortolioMapper*, and calls its *get* method indicating only the exact path and file name. The class argument is no longer needed, since the type of object is already hardwired in the *PortfolioMapper* code.

After getting a reference to the newly created *Portfolio* object, the controller calls its *validate* method so that node's x and y coordinates as well as other business rules get validated. If no major errors were found, controller calls the *Portfolio*'s method *genTemplate* in order to customize and generate the new Layers template file.

The *genTemplate* method of the *Portfolio* object is presented in Figure 4.5 on page 74. This method consists of updating the Layers base template to reflect the new Portfolio's artwork specification. After the template has been customized, it is physically saved on hard disk under a new name. To finalize, the *genTemplate* method creates an instance of the *NetworkMap* object initializing it with a 'demo' parameter. The 'demo' parameter indicates that all nodes are to be activated so that the technical illustrator can visualize the final result. The new template name and the *NetworkMap* object's reference are returned to the controller.

1. Portfolio object gets an instance of the PersistenceFacade object and calls its get() method to have the LayersTemplate object instantiated.

2. Portfolio sends the appropriate information about the new template that needs to be created to the customizeTempl() method of the LayersTemplate object.

3. The new template is physically saved by the LTMapper.

4. Portfolio instantiates the NetworkMap object and initializes it with a 'demo' parameter. This means all nodes and menu options need to be activated

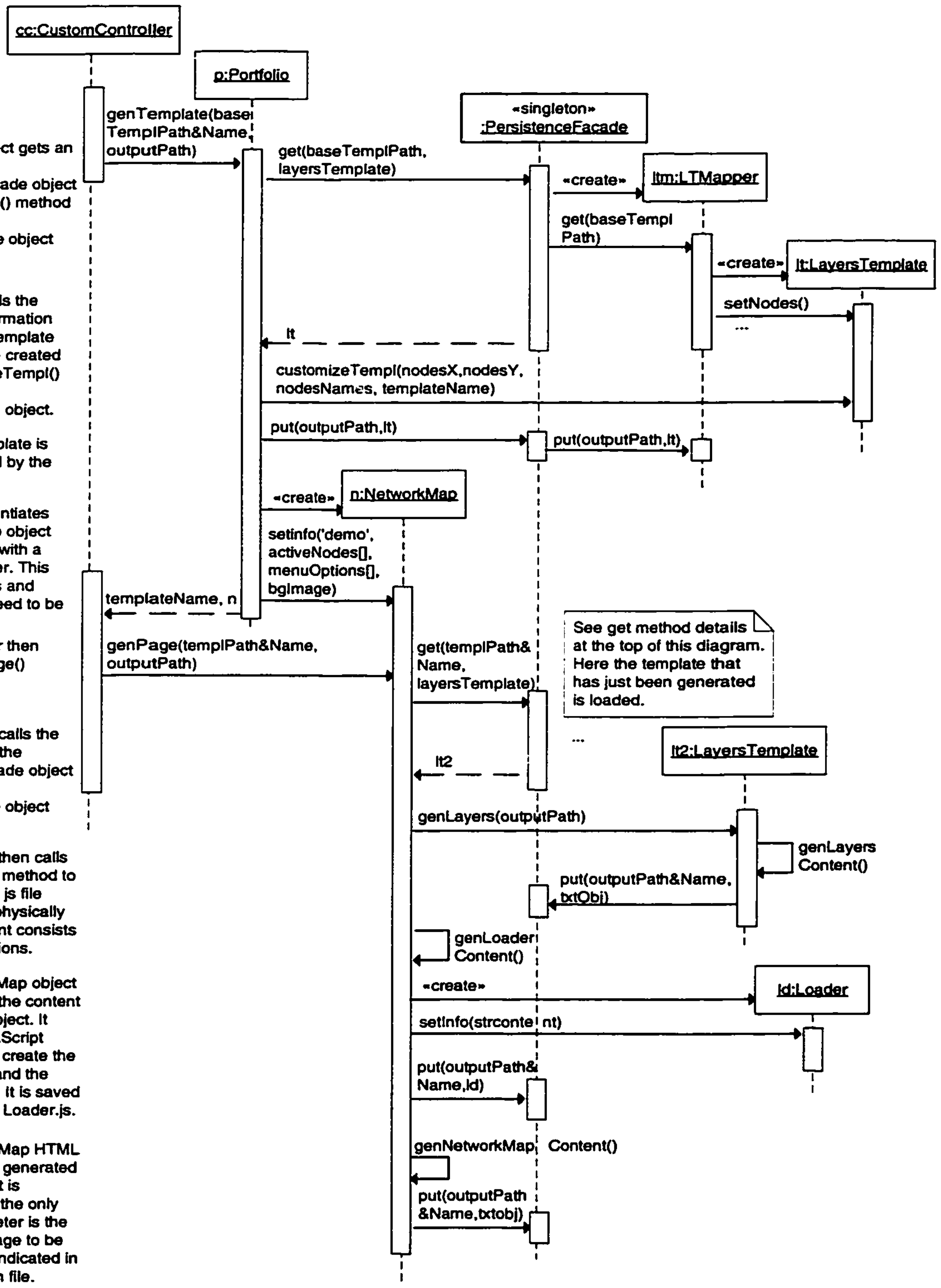
5. The controller then calls the genPage() method of the NetworkMap.

6. NetworkMap calls the get() method of the PersistenceFacade object to have the LayersTemplate object instantiated.

7. NetworkMap then calls the genLayers() method to have the Layers js file generated and physically saved. Its content consists of DHTML functions.

8. The NetworkMap object then generates the content of the Loader object. It consists of JavaScript function calls to create the Welcome Text and the release menus. It is saved under the name Loader.js.

9. The NetworkMap HTML page content is generated as well. Content is predefined and the only variable parameter is the background image to be used, which is indicated in the specification file.



cc:CustomController
p:Portfolio
«singleton» :PersistenceFacade
ltm:LTMapper
lt:LayersTemplate
n:NetworkMap
lt2:LayersTemplate
ld:Loader
genTemplate(baseTemplPath&Name, outputPath)
get(baseTemplPath, layersTemplate)
«create»
get(baseTemplPath)
«create»
setNodes()
...
lt
customizeTempl(nodesX,nodesY, nodesNames, templateName)
put(outputPath,lt)
put(outputPath,lt)
«create»
setinfo('demo', activeNodes[], menuOptions[], bgImage)
templateName, n
genPage(templPath&Name, outputPath)
get(templPath& Name, layersTemplate)
See get method details at the top of this diagram. Here the template that has just been generated is loaded.
...
lt2
genLayers(outputPath)
genLayers Content()
put(outputPath&Name, txtObj)
genLoader Content()
«create»
setInfo(strcontent)
put(outputPath& Name,ld)
genNetworkMap Content()
put(outputPath &Name,txtobj)

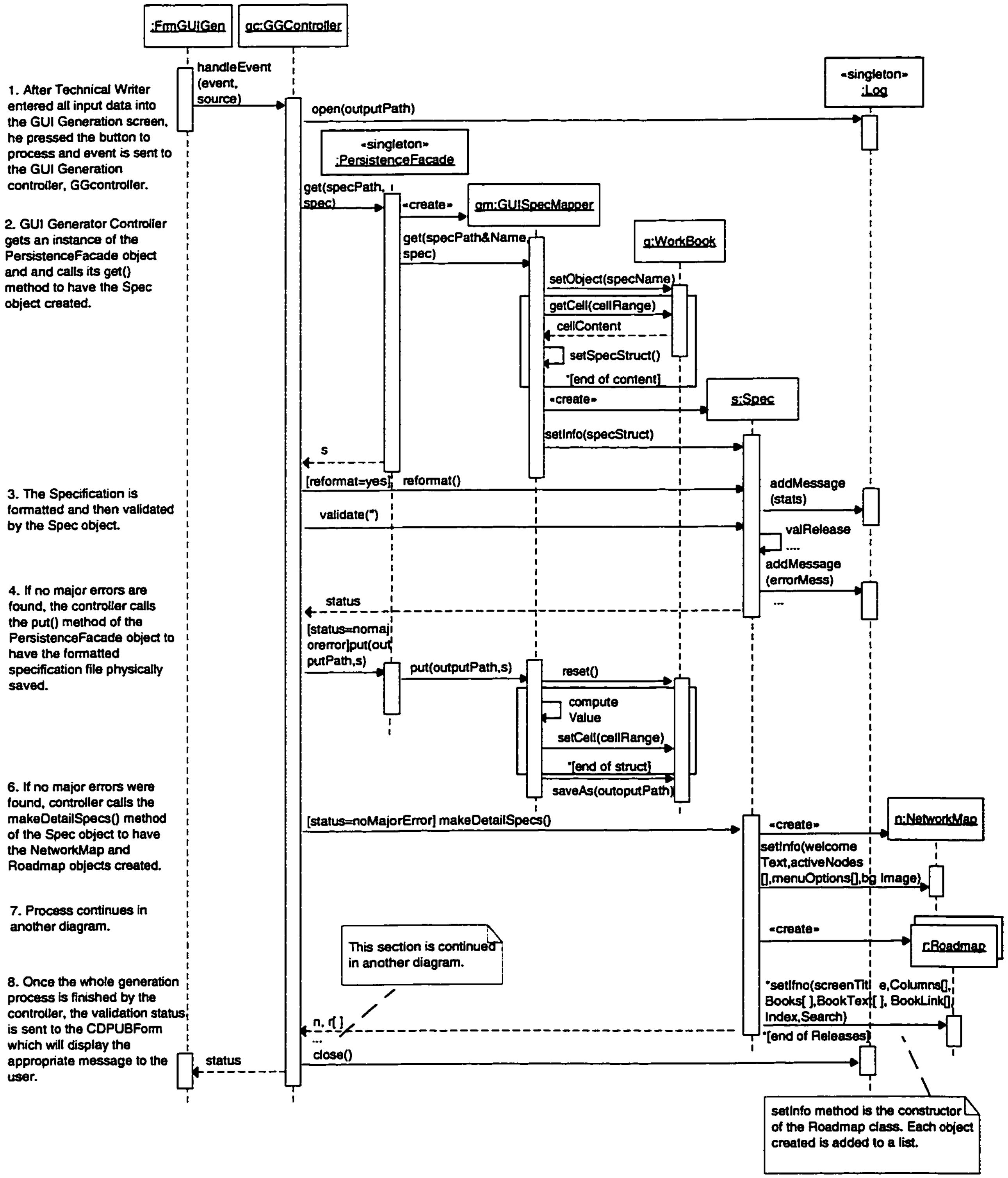
Figure 4.5 Sequence Diagram for GUI Customization Use Case, Part II.

After that, Controller calls the NetworkMap object's method *genPage* also presented in Figure 4.5. The new instantiation of the *LayersTemplate* object might seem as task duplication, since it has been previously instantiated on the *genTemplate* method. However, separation of concerns as well as code reuse were behind this design decision and we will see on the next use case the advantages of this implementation. The method *genLayers* of the *LayersTemplate* object generates the content of the JavaScript Layers file and physically saves it by using the services of the singleton *PersistenceFacade*. After this, the *NetworkMap* creates the *Loader* object and sets its content, which consists of calls to the JavaScript menu functions contained in the Layers JavaScript file. *Loader* is also physically saved on hard disk. Finally, the *NetworkMap* object generates the content of the Network Map HTML page, and physically saves it.

After control returns to the *CustomController* object, as shown on Figure 4.4 on page 72, the latest calls the *close* method of the *Log* object to have it physically saved to disk. The controller's method *handleEvent* ends by returning the validation status to the *FrmGUICustom* object which in turn presents an appropriate message to the user.

4.3.2 GUI Generation Publishing Option

Another essential service the CD-ROM PUB system provides is the GUI Generation use case. On page 76, Figure 4.6 presents the sequence of messages that get exchanged between the classes participating in the GUI Generation use case, Part I. Once the actor of this use case, the technical writer, presses the process button the *FrmGUIGen* object sends the message *handleEvent* to the *GGController* oject.



1. After Technical Writer entered all input data into the GUI Generation screen, he pressed the button to process and event is sent to the GUI Generation controller, GGcontroller.

2. GUI Generator Controller gets an instance of the PersistenceFacade object and and calls its get() method to have the Spec object created.

3. The Specification is formatted and then validated by the Spec object.

4. If no major errors are found, the controller calls the put() method of the PersistenceFacade object to have the formatted specification file physically saved.

6. If no major errors were found, controller calls the makeDetailSpecs() method of the Spec object to have the NetworkMap and Roadmap objects created.

7. Process continues in another diagram.

8. Once the whole generation process is finished by the controller, the validation status is sent to the CDPUBForm which will display the appropriate message to the user.

Figure 4.6 Sequence Diagram for the GUI Generation Use Case, Part I.

GGController object gets an instance of the PersistenceFacade singleton class and calls its *get* method with spec class as a parameter, to have the *Spec* object materialized. The *Spec* object captures the GUI specification instructions previously captured in a spreadsheet by the technical writer. The *PersistenceFacade* class will in turn create an instance of the appropriate mapper object, in this case the *GUISpecMapper*, and calls its *get* method indicating only the exact path and file name. After getting the reference to the *Spec* object, the *GGController* calls its *reformat* method so that the specification text content is reformatted accordingly to the business rules. Statistics about quantity of Roadmap pages, quantity of book links per page, spreadsheets processed, and others are generated and added to the log. After the reformatting is applied, the *validate* method of the *Spec* object is called to have the specification validated. If no major errors were found, *GGController* makes sure the reformatted *Spec* object gets physically saved. This is done in order to offer the user the choice of manually changing the reformatted spreadsheet, since the user might want to change the way the formatting was done in certain cases. The user can indicate to the system that the spreadsheet is already reformatted, by setting the appropriate parameter in the customization form, so that no formatting is automatically done.

If no major errors were found, *GGController* then calls the *makeDetailSpecs* of the *Spec* object. This method organizes the global specification into two separate objects, the *NetworkMap* object and the *Roadmap* collection object. The *genPage* method calls are illustrated on page 78, Figure 4.7. As the figure states, the *genPage* method of the *NetworkMap* object has been previously explained in the GUI Customization use case sequence diagram, Part II on Figure 4.5 on page 74. Code reuse is achieved. The *genPage*

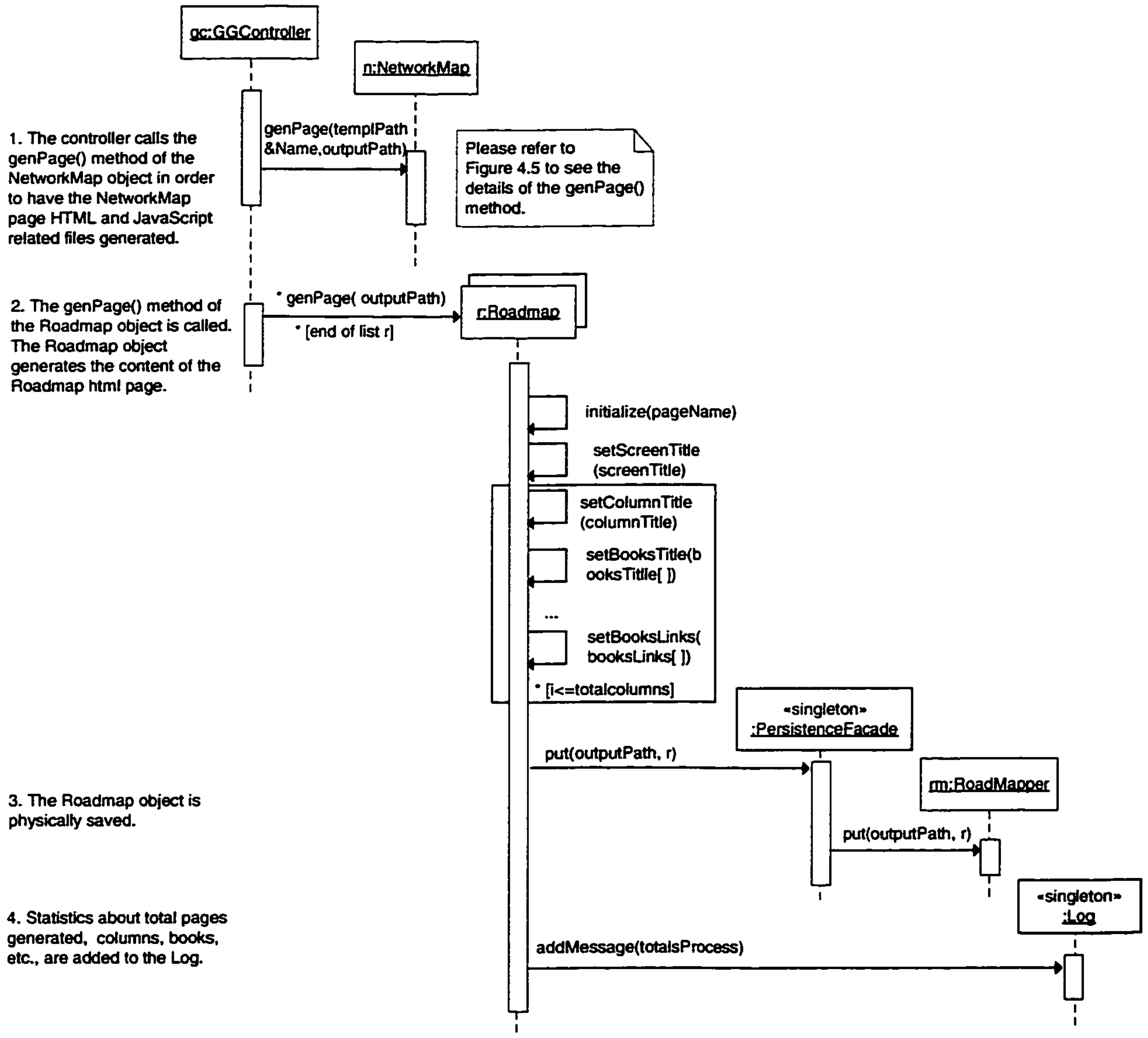

gc:GGController
n:NetworkMap
1. The controller calls the genPage() method of the NetworkMap object in order to have the NetworkMap page HTML and JavaScript related files generated.
genPage(templPath &Name,outputPath)
Please refer to Figure 4.5 to see the details of the genPage() method.
2. The genPage() method of the Roadmap object is called. The Roadmap object generates the content of the Roadmap html page.
* genPage(outputPath)
* [end of list r]
r:Roadmap
initialize(pageName)
setScreenTitle (screenTitle)
setColumnTitle (columnTitle)
setBooksTitle(b ooksTitle[ ])
...
setBooksLinks(booksLinks[ ])
* [i<=totalcolumns]
«singleton» :PersistenceFacade
put(outputPath, r)
rm:RoadMapper
put(outputPath, r)
3. The Roadmap object is physically saved.
«singleton» :Log
4. Statistics about total pages generated, columns, books, etc., are added to the Log.
addMessage(totalsProcess)

Figure 4.7 Sequence Diagram for the GUI Generation Use Case, Part II.

method of each of the members of the *Roadmap* collection is called. This method consists of setting the values of the data structures that represent the content tags, like page name, roadmap columns titles, book title, etc. These tags are later on used as input for the Roadmap Applet previously designed in chapter 3. Once the data structures are set, the

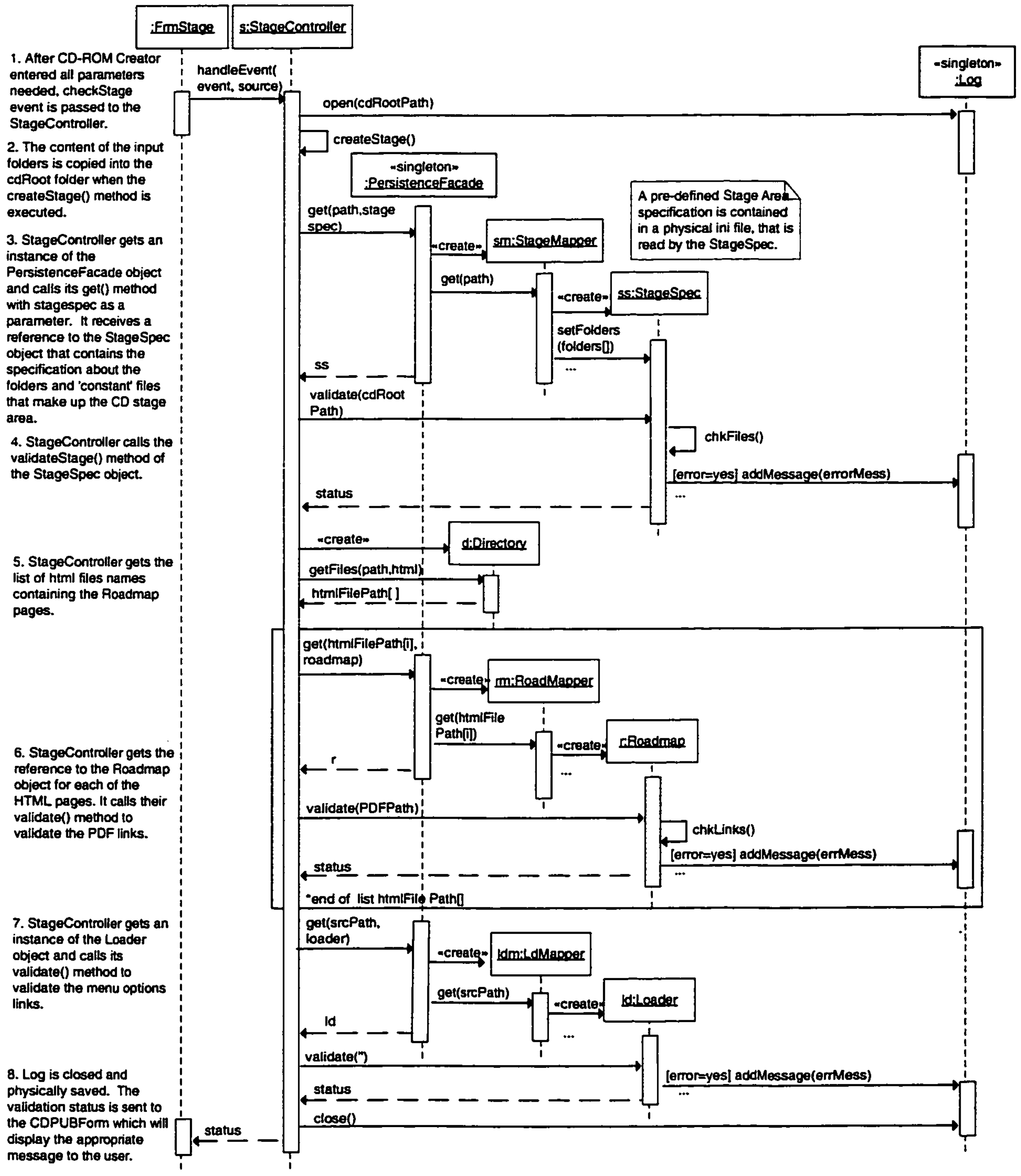
Roadmap object calls the *put* method of the *PersistenceFacade* object with roadmap class as a parameter. The Roadmap HTML page gets physically saved into disk. As well, and for each processed *Roadmap* object in the collection, statistics are generated and added to the log.

Once control returns to the *GGController* object, as shown on previous Figure 4.6, the controller closes the log and returns the status to the *FrmGUIGen* object.

4.3.3 CD-ROM Stage Area Creation Publishing Option

On page 80, Figure 4.8 presents the sequence diagram for the CD-ROM Stage Area Creation use case. After the CD-ROM creator chooses this option from the CD-ROM PUB system, the *StageController* object is created by the *FrmStage* object. After setting the Log location, it performs its private method *createStage* that consists of copying all the files from the src folders to the root folder of the CD-ROM virtual partition indicated by the user. After that, the *StageSpec* object is materialized by the *PersistenceFacade*. It contains the specification of the constant folders and files that must be included in the stage area of the CD-ROM GUI application. After getting the reference to the *StageSpec* object, the controller calls its *validate* method with the cd root path as an argument. This method compares the actual content of each of the src folders against the stage specification information previously materialized into the *StageSpec* object. The *Log* object is updated with any error messages that might apply.

After the validation of the stage constant file structure, the controller gets the list of the Roadmap HTML files stored in the src folder. It then iterates through this list, to get the instance of the corresponding *Roadmap* object and call its *validate* method to have all



1. After CD-ROM Creator entered all parameters needed, checkStage event is passed to the StageController.

2. The content of the input folders is copied into the cdRoot folder when the createStage() method is executed.

3. StageController gets an instance of the PersistenceFacade object and calls its get() method with stagespec as a parameter. It receives a reference to the StageSpec object that contains the specification about the folders and 'constant' files that make up the CD stage area.

4. StageController calls the validateStage() method of the StageSpec object.

5. StageController gets the list of html files names containing the Roadmap pages.

6. StageController gets the reference to the Roadmap object for each of the HTML pages. It calls their validate() method to validate the PDF links.

7. StageController gets an instance of the Loader object and calls its validate() method to validate the menu options links.

8. Log is closed and physically saved. The validation status is sent to the CDPUBForm which will display the appropriate message to the user.

A pre-defined Stage Area specification is contained in a physical ini file, that is read by the StageSpec.

Figure 4.8 Class Diagram for the CD-ROM Stage Area Creation Use Case.

the book's PDF links validated. The log is updated with the appropriate messages and a validation status is returned with each list's iteration. Finally, the last stage validation is performed by getting an instance of the *Loader* object and calling its *validate* method. *Loader* validates that all menu links references to pages exist. It updates the log with the appropriate messages and returns a validation status. To finish the process, *StageController* closes the log and returns a validation status to the *FrmStage* object.

4.4 Architecture: Component Diagram

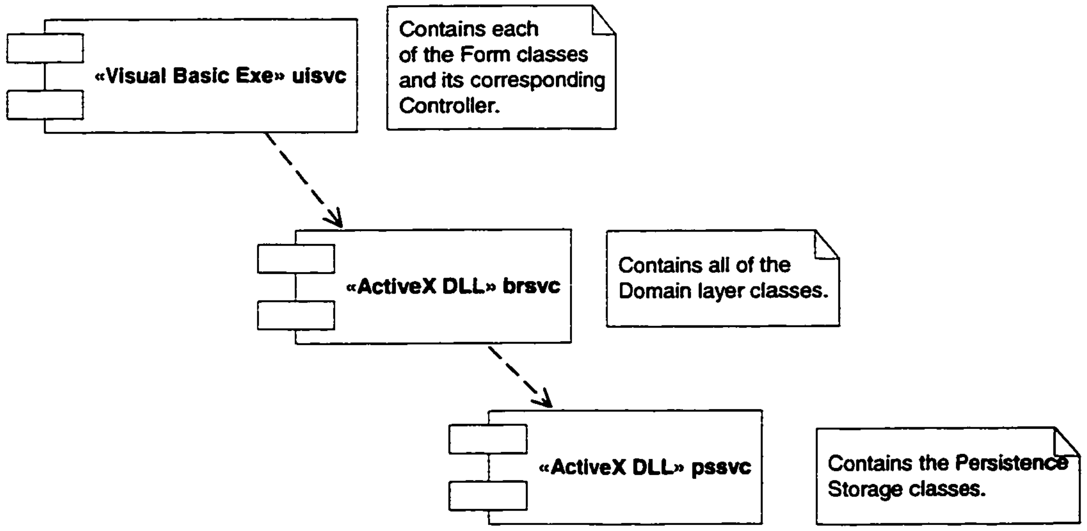

«Visual Basic Exe» uisvc
Contains each of the Form classes and its corresponding Controller.
«ActiveX DLL» brsvc
Contains all of the Domain layer classes.
«ActiveX DLL» pssvc
Contains the Persistence Storage classes.

Figure 4.9 CD-ROM PUB System Component Diagram.

Figure 4.9 above, shows the component diagram for the CD-ROM PUB system. The component *uisv,* User Interface Services, maps to the logical Presentation layer; *uisvc.exe* will be a standard Visual Basic EXE project and will contain all the forms and its associated controllers. This component depends on the next component, *brsvc,* Business

Rules Services. *Brsvc.dll* will be an ActiveX DLL in process, which means that it will run in the project space of the *.exe*. It will contain all the domain classes and depends on the next Persistence Storage component, *pssvc. Pssvc.dll* will be made into an ActiveX DLL in process component as well. It will contain the *PersistenceFacade*, all the mapper classes, and their helper classes. [Reed].

CHAPTER 5 CONCLUSION

We have seen how the documentation needs of both technological companies and their customers can be satisfied with the EDSTDL system solution. The qualities of this solution satisfy both users' requirements for a low budget and fast solution as well as for a highly searchable and navigable uniform visual interface. The present economic conditions played an important role in the conception of this solution. The solution needed to fit into the companies' legacy publishing systems; otherwise its cost would have been too elevated.

The complete solution, which consists of the two subsystems CD-ROM GUI and CD-ROM PUB, had to be carefully designed in order to minimize the interdependency between these two subsystems. The CD-ROM GUI's variable components were clearly identified and separated from the static ones, in order to automate the generation of the formers. Through the use of UML, object-oriented design techniques, and the application of software design patterns, the conception of a software system that could be easily extended and maintained was achieved. The classes presented here represent the base of a class library that can be reused in the future in other applications.

From the documentation readers' point of view the presence of searchable PDF files and a browser in the final solution, means interacting with visual interfaces already familiar and that have become of universal use. Hence, the users' learning curve is reduced to a minimum.

The decision of making the visual interface browser-based had the non-negligible advantage of code reuse on the system's future Web implementation. However, this decision brought some constraints to the system. The use of Sun's Java virtual machine became mandatory since programming for so many different flavours of each existent browser's Java would be an almost impossible task to achieve. Similarly, the need for a Security Certificate to sign the applet arose, and with this the present limitations of the available code signing certificates, whose maximum period of validity is only a year. All that having said, it could be useful in the future, to make a conversion of the CD-ROM GUI Presentation layer from browser-based to stand alone Java application. This conversion would be very straightforward; the applet already being a kind of mini-application and the Network Map page having an object based design. Furthermore, the system would benefit from moving the Network Map page from the rather unstable JavaScript environment to the more stable Java programming language.

Future work can certainly be done in order to add Web and Server-based functionalities to the application. As well, the automatic generation of the GUI specifications files, in a XML format instead of Excel spread sheets inputted manually, could be implemented by extracting the books' title and summary directly from the Frame Maker or PDF source files.

The use of the UML for the modeling of the system was an enriching experience. The solution's UML model can be directly connected to different programming languages. In the future, any developer can interpret this model in a unique way, since each UML symbol and notation has well-defined semantics behind it.

To conclude, the EDSTDL system solution contributes in an original and effective way to the fulfillment of the industry's need for publishing technical documentation in a universal searchable electronic format within restricted economic and time deadlines.

BIBLIOGRAPHY

Adobe Systems Incorporated. "Adobe® PDF Workflows for Print Production". BCXXXX 01/01. Retrieved February 8, 2003 from <http://www.adobe.com/print/prodzone/pdfs/PDFWorkflow.pdf>.

---. "Adobe® Acrobat® 5.0 User Guide for Windows® and Macintosh". 2001. CD-ROM. March 3, 2001.

---.Adobe Frame Maker Home Page. 2002. Retrieved September 01, 2002 from <http://www.adobe.com/products/framemaker/keyfeature9.html>.

Barret, Dan. Essential JavaScript™ for Web Professionals. Ed. John Neidhart. 2nd. ed. Upper Saddle River, NJ: Prentice Hall PTR, 2003.

Booch, Grady, James Rumbaugh, and Ivar Jacobson. The Unified Modeling Language User Guide. Reading, Massachusetts: Addison-Wesley, 1999.

Buschmann, Frank, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. Pattern-Oriented Software Architecture: a System of Patterns. Chichester, New York: John Wiley & Sons, 1996.

Conallen, Jim. Building Web Applications with UML Second Edition. 2nd. ed. Boston: Addison-Wesley, 2003.

Constantine, Larry L. and Lucy A. D. Lockwood. Software for use: a practical guide to the models and methods of usage-centered design. Reading, Massachusetts: Addison-Wesley, 1999.

Eidahl, Loren D., et al. Using Visual Basic® 6. Ed. Sharon Cox. Platinum ed. Indianapolis: Que Publishing, 1999.

Goodman, Danny. JavaScript® Bible. Ed. Nel Romanosky. Gold ed. Indianapolis: Hungry Minds, Inc., 2001.

Hardee, Martin et al. "What is Java™ Technology". 2002. Retrieved October 01, 2002 from <http://java.sun.com/java2/whatis/>.

Holzner, Steven. Java Black Book. Scottsdale, Arizona: Coriolis Technology Press, 2000.

Larman, Craig. Applying UML and Patterns: an Introduction to Object-Oriented Analysis and Design and the Unified Process. Upper Saddle River, NJ: Prentice Hall PTR, 2002.

Microsoft Corporation. "MSDN Library Visual Studio 6.0". 1995-2000. CD-ROM. December 15, 2002.

Reed, Paul R. Jr. Developing Applications with Visual Basic and UML. Reading, Massachusetts: Addison-Wesley, 2000.

Roman, Steven, Ron Petrusha, and Paul Lomax. VB.NET Language in a Nutshell. Ed. Ron Petrusha. 2nd. ed. Beijin: O'Reilly, 2002.

Search Tools Consulting. "Searching PDF files". June 19, 2002. Retrieved September 09, 2002 from <http://www.searchtools.com/info/pdf.html>.

Sun Microsystems, Inc. "How RSA Signed Applet Verification Works in Java Plug-in".2002. Retrieved August 28, 2002 from <http://java.sun.com/j2se/1.4.1/docs/guide/plugin/developer_guide/rsa_how.html>.

Sun Microsystems, Inc. "Frequently Asked Questions – Java Security".2001. Retrieved July 26, 2001 from <http://java.sun.com/sfaq/>.

Weber, Joseph L. Using Java™ 2 Platform. Ed. Tim Ryan. Special ed. Que Publishing, 1998.