Automating Software Deployment

A Ph.D. Thesis by

Benchiao Jai

New York University
Department of Computer Science

Acknowledgements

There is a phrase widely used in Chinese fortune-telling: “From south-east come people of great help.” In my limited life experience, helpful people seem to come from all directions. There are so many people who helped me in so many ways, the appreciation I can express on paper can only show the tip of the iceberg.

First and foremost, I thank my advisor Alan Siegel for guiding me, both in school and in real life, through quite a few whimsical years of my life. I often find myself quoting his words of wisdom to my junior fellows. I also thank Allan Gottlieb, Eric Freudenthal, and other colleagues in the UltraComputer Laboratory for some very stimulating research experience; Arthur Goldberg for showing me the way to the world of GroupWare and the Internet; Malcolm Harrison, Robert Dewar, Benjamin Goldberg, Edmond Schoenberg, and other knowledgeable minds in the Language Laboratory for broadening my views; Richard Cole, often through short and wise comments, setting a vivid example as a research scholar; my dear friends Charlie Repetti, Nai-Wei Hsue, Hseuming Chen and David Solomon for feeding me mental stimuli and connections to industry; and last but not least, Peter Barnett for teaching me the responsible perspectives towards many things.

My parents Shing-Jeng Jai and Chio-Jane Hu brought me into this world and nurtured me through difficult times. My wife Angela kept me motivated. Without their love, I would not have lasted so long in this endeavor. My brothers Robin and Jackson are always setting good examples for me to follow. My admiration toward them can not be described by words.

Title: Automating Software Deployment
Author: Benchiao Jai
Advisor: Alan Siegel

Abstract

The work users do with an application can be divided into actual work accomplished using the application and overhead performed in order to use the application. The latter can be further partitioned based on the time at which the work is performed: before (application location and delivery), during (installation) and after (upgrade) the installation of the application. This category can be characterized as the software deployment overhead. This thesis presents a component architecture RADIUS (Rapid Application location, Delivery, Installation and Upgrade System) in which applications can be built with no software deployment overhead to the users. An application is deployed automatically by simply giving the user a document produced by the application. Furthermore, the facilities in RADIUS make the applications self-upgrading. In the end, the users perform no deployment overhead work at all.

The conventional way of using an application is to install the application first, then start using documents of the application. The object-oriented programming (OOP) paradigm suggests that this order should be reversed: the data should lead to the code. However, almost all software fails to meet this model of design at the persistence level. While modern software often use OOP at the program level, the underlying operating systems do not support OOP at the document/file level. OOP languages use pointers to methods to indicate what operations can be performed on the objects. We extend the idea to include "pointers to applications." Each document has an attached application pointer, which is read by RADIUS when the document is opened. This application pointer is then used to locate and deliver the application module necessary for the document.

RADIUS is designed to be compatible with existing technologies and requires no extensions to either programming languages or operating systems. It is orthogonal to programming tools, is language-independent and compatible among operating systems, and consequently does not impose limitations on which environments the developers can use. We illustrate the implementations for the two most popular platforms today – C++ on Windows, and Java. RADIUS is also orthogonal to other component systems such as CORBA or COM and is easy to integrate with them.

Table Of Contents

Acknowledgements .... **iii**

Abstract .... **iv**

List Of Figures .... **vii**

Chapter 1: Introduction **1**

1.1. Motivation 1

1.2. Design Principles 4

1.3. Extending the Object-Oriented Programming Paradigm 5

1.4. Related Work 7
- 1.4.1. Installation Managers 7
- 1.4.2. Netscape Plug-in Finder and SmartUpdate 7
- 1.4.3. Marimba Castanet 8
- 1.4.4. Java Applications and Applets 8
- 1.4.5. Operating System Facilities 9
- 1.4.6. Component Systems 9
- 1.4.7. Miscellaneous Application-Specific Code 10
- 1.4.8. Summary 10

1.5. Thesis Organization 10

Chapter 2: The RADIUS Application Framework **12**

2.1. The RADIUS Data Structure and Algorithm 12

2.2. The Application Server 15

2.3. Storage Space Management 16

2.4. Wrapping Existing Applications into RADIUS 17

2.5. Summary 17

Chapter 3: RADIUS Implementations **19**

3.1. The RADIUS Programming Interface 19
- 3.1.1. The RADIUS Programming Interface in Java 19
- 3.1.2. The RADIUS Programming Interface in C++ 20

3.2. The Application Loader 22
- 3.2.1. The Application Loader in Java 23
- 3.2.2. The Application Loader in C++ 28
- 3.2.3. Naming Application Files 31

3.3. The Object Factory 31
- 3.3.1. The Object Factory in Java 32
- 3.3.2. The Object Factory in C++ 33

3.4. The Object Browser....33

Chapter 4: Application – Programming in Presentation....36

4.1. Motivation....36
4.1.1. Programming Models....36
4.1.2. Related Work....37

4.2. The Basic Component....38
4.2.1. The Data Structure....38
4.2.2. The Data Methods....39
4.2.3. The Visual Interface....40
4.2.4. The Event Interface....42
4.2.5. The Scripting Interface....43
4.2.6. Viewing and Editing PIP Objects....44

4.3. Standard PIP Components....46
4.3.1. The PIP_Text Class....47
4.3.2. The PIP_Picture Class....47
4.3.3. The PIP_Page Class....49
4.3.4. The PIP_Presentation Class....50
4.3.5. The PIP_Line, PIP_Shape and PIP_Polygon Classes....50
4.3.6. The PIP_Timer Class....52
4.3.7. The PIP_Media Class....52
4.3.8. The PIP_Hotspot Class and the PIP_Button Class....53
4.3.9. The PIP_OLE Class....54
4.3.10. The PIP_Engine Class....54
4.3.11. The PIP_Table and PIP_Chart Classes....55
4.3.12. The PIP_Database Class....55

4.4. PIP Script Programs....55
4.4.1. Security for PIP Objects....57

4.5. Summary....58

Chapter 5: Conclusion....59

5.1. Application Areas of RADIUS....59

5.2. Future Directions....60
5.2.1. Generalizing RADIUS....60
5.2.2. Integrating the Object Browser....60
5.2.3. The Storage and Exchange Format....61
5.2.4. COM/CORBA Interfaces....61
5.2.5. Software Licensing....61
5.2.6. Security and Authentication....62
5.2.7. RADIUS Applications as Mobile Code....62

BIBLIOGRAPHY....63

List Of Figures

Figure 1: Software Activities and Participants 1
Figure 2: Analogy between RADIUS and OOPL 6
Figure 3: A Sample RADIUS Document 12
Figure 4: The RADIUS Process 12
Figure 5: Key RADIUS Methods (in Java Notation) 14
Figure 6: Roles of RADIUS Documents 14
Figure 7: Additional RADIUS Methods (in Java Notation) 15
Figure 8: RADIUS Interface in Java 19
Figure 9: Standard RADIUS Application in Java 20
Figure 10: Definition of the RADIUS Base Class in C++ 21
Figure 11: Sample RADIUS Application in C++ 22
Figure 12: Key Portion of Application Loader in Java 23
Figure 13: Dependency among Classes from Different Hosts 26
Figure 14: Caveat in Using the Application Loader in Java 27
Figure 15: Code in Object Browser to Encapsulate the Application Loader 27
Figure 16: Simple Meta-Object `Class` in C++ 28
Figure 17: Definition of Application Loader in C++ 29
Figure 18: Sample Windows Resource Script Specifying DLL Dependency 31
Figure 19: Object Factory in Java 32
Figure 20: Definition of Object Factory in C++ 33
Figure 21: RADIUS Object Browser in Java 33
Figure 22: RADIUS Object Browser in C++ 33
Figure 23: A Directory Window in RADIUS – Java Version 34
Figure 24: A Directory Window in RADIUS – C++ Version 34
Figure 25: Code to Display a Document 35
Figure 26: Text Application in Java 35
Figure 27: Text Application in C++ 35
Figure 28: Picture Application in Java 35
Figure 29: Picture Application in C++ 35
Figure 30: A Typical Sequential Program 36
Figure 31: A Typical Event-Driven Program 36
Figure 32: A Typical Presentation Style Program 37
Figure 33: PIP File Structure 38
Figure 34: Data Methods in `PIP_Object` 39
Figure 35: Visual Methods in `PIP_Object` 40
Figure 36: Event-related Methods in `PIP_Object` 42
Figure 37: Other PIP Event Types 43
Figure 38: The `PIP_Script_Engine` interface 44
Figure 39: PIP Event Model 44
Figure 40: Viewing and Editing Methods in `PIP_Object` 45
Figure 41: Standard Editing Window for `PIP_Object` 46
Figure 42: Customized Standard Editing Window for `PIP_Picture` 46
Figure 43: A `PIP_Text` Object 47
Figure 44: Editing a `PIP_Text` Object 47

Figure 45: A `PIP_Picture` Object........48
Figure 46: Editing a `PIP_Picture` Object........48
Figure 47: PIP Page Designer........49
Figure 48: A Curved Frame for a `PIP_Line` Object........50
Figure 49: Editing a `PIP_Line` Object........50
Figure 50: Editing a `PIP_Shape` Object........51
Figure 51: Editing a `PIP_Polygon` Object........51
Figure 52: Editing a `PIP_Media` Object........53
Figure 53: Editing a `PIP_Button` Object........53
Figure 54: Editing a `PIP_OLE` Object Encapsulating an Excel Spreadsheet........54
Figure 55: PIPScript Grammar Rules........56
Figure 56: Sample PIPScript Program........57
Figure 57: RADIUS Application Areas........59
Figure 58: Current Status of RADIUS Implementations........60

Chapter 1: Introduction

Software deployment issues originate from the field of software engineering. This thesis presents a solution using a technique commonly employed in object-oriented programming languages. The implementation of the solution makes use of the latest data communication technologies and is done at a level that can be considered part of the operating systems. By combining the knowledge from a diversified set of fields, we were able to find and implement the solution in a clean way with very little coding.

This dissertation describes the RADIUS (Rapid Application location, Delivery, Installation and Upgrade System) application framework. Two implementations are presented in detail and sample applications are illustrated. Useful software systems built from the sample applications are also demonstrated. In this chapter, we review the motivation, the design principles, the solution, related work, and the organization of the rest of this thesis.

1.1. Motivation

Using software has become a daily activity of technologically advanced societies on this planet. Between the point where a software is available and the point where the software is usable, several steps have to be performed:

- (Software Location.) The user first has to find the software. For example, when the user downloads a PDF file, he or she needs to find Acrobat Reader™. When a postscript file is received through email, a postscript viewer has to be located. Even after the application is installed, the application location process is still performed in the background by the operating system every time a document is opened.
- (Software Delivery.) The user has to get a copy of the software. Some software can be downloaded over the Internet, some can be purchased in stores or through mail order, and some custom-made applications are copied over an internal network within organizations.
- (Software Installation.) The software has to be loaded on the user's computer. Computers that do not have application software stored on them but may have applications executed on them, such as diskless workstations on a LAN or terminals, are considered as a part of a larger computer for the scope of this thesis.
- (Software Upgrade.) If there is a newer version of a software, sometimes the users will like to use the newer version instead of the old one, and therefore the previous steps need to be repeated and some coordination tasks need to be performed.

We call these collective steps the software deployment process.

Activity	Typical Participants
1. Software specification and analysis	Project manager, system analyst
2. Software design	System analyst, programmer
3. Implementation, testing	Programmer, quality manager
4. Documentation	System analyst, programmer, technical writer
5. Delivery	Project manager or software vender, user
6. Installation, upgrade	User, technical support
7. Training, operation	User, technical support

Figure 1: Software Activities and Participants

Software activities contain many parts performed by different groups of people. The figure above shows a typical partitioning of software activities. In the past few decades significant effort has been devoted to reducing the cost of developing (items 1 through 4) and operating software (item 7). Until recently, little attention has been paid to reducing the cost of deploying software (items 5 and 6) [16]. We conjecture from our experience that the software deployment issues did not gain recognition due to the following factors:

- Compared against the thousands or millions of person-hours spent on developing the software, the few person-hours or person-days spent on deploying the software is just a tiny fraction of the total time and cost.
- The time and cost associated with the users' part is relatively small or not visible in terms of dollar amount. The users' expense was not in the accounting formula.
- Any effort to make the deployment process easy will be performed by developers, who tend to put higher priority on their own needs, which is to get the project done. It is a well-known fact that software projects are more likely to be completed late than on time, therefore developers are more likely to work on making their own lives easier than making the users' lives easier.
- Developers have a mentality that subconsciously says that once the product is delivered, their duty is over and they don't want to spend any more time on it.

However, the situation becomes different as the computing environment evolves. Today's computing environment is globally connected, distributed, shared, dynamic and changing rapidly. We are witnessing an evolution where the era of "personal computer" is being replaced by the era of "the network is the computer" (Sun, 1988) [13]. Studies on TCO (Total Cost of Ownership) [8, 37] of computers have indicated that software deployment tasks actually incur a notable cost to the users, therefore the users are beginning to demand more development effort to reduce their burden. Several new trends have significantly increased the visibility of software deployment tasks and led to this awakening of the users:

A. Software reuse technologies, such as component architectures [26, 29, 45], are reducing the number of applications that have to be developed from scratch.
B. RAD (Rapid Application Development) tools [5, 38, 40] are reducing the development effort for each application.
C. Easy-to-use programming tools are encouraging the average programmers to develop applets (tiny applications).
D. The number of software users has increased dramatically.
E. The "release early, release often" practice [39] is becoming commonplace, therefore is increasing the number of software upgrades.
F. The price of computer hardware has dropped substantially, therefore encouraged hardware upgrades.

The total effort spent on deploying software is proportional to 1) the number of software applications to be deployed (trend C), 2) the number of users using the software (trend D), 3) the number of releases of software (trend E) and 4) the number of times the same software has to be deployed (trend F). Combined with the relative slowing down of the growth in software development effort (trends A and B), one can easily visualize the burgeoning demand on solutions to software deployment issues. A few example situations can help stress the importance of cutting down the time and effort involved in, or even totally automating, the software deployment process:

- (Insufficient instruction.) Five hundred copies of a powerful mathematics education system are delivered to the public schools in some state. The installation instruction is seven pages long. Two semesters later only twenty-five copies are in use because most elementary school teachers were not able to get everything installed and configured correctly.
- (Incoherent operations.) A large corporation is upgrading its standard word processor system. Eight technical support technicians spend two months working through four thousand workstations installing the upgrade, but missed seventeen notebook computers used by some travelling salespeople. Six months later some of the salespeople lost big accounts because they were not able to read a very important memo from the CEO.
- (Replacement of legacy configuration.) A (non-computer science) professor is using thirty-eight different software packages to conduct a research. The research group decides to upgrade all the computers to the new 1-GHz model. The professor can't figure out which files to move over since most of the packages were installed by various research assistants and hardware/software venders. The new machines are therefore sitting in a corner collecting dust.
- (Failure to locate software.) The same professor uses a very powerful proprietary software package for processing research data. When the research result is sent out for peer review, five out of five reviewers do not know how to read and verify the data. One hundred sixty-two email messages, fifty-eight FTP sessions and four months later, the reviewers finally figured out what the professor was doing.

We are not claiming that no one has noticed the problem. Indeed, a vast amount of time and money has been spent on creating installation and upgrade packages (on the developers' end) and setting up and maintaining software configurations (on the users' end). We can see that there is a huge demand by observing:

- The colossal success of InstallShield™. InstallShield simplifies the processes of both building the deliverable software package and installing the software. It transforms a major part of the process of creating an installation package into a checklist with fill-in blanks and some minor scripting, and turns the installation process into mostly a set of select-and-go steps. At one time, the company that designed InstallShield claimed that seventy-five percent (75%) of the software in the world is installed by InstallShield.
- The expanding size of corporate technical support department. As the number of software packages and users grow, more staff is needed to handle the construction and maintenance of software configurations. According to a study [54] by International Data Corp., seventy-five percent (75%) of the lifetime cost of a computer is spent on staffing while only nine percent (9%) is on software and sixteen percent (16%) on hardware.
- The rapid growth of software sales and delivery over the Internet [16]. Instead of having to go from computers to secondary storage media to stores then back to the computers, software packages can go directly from computers to computers.

These recent developments are dedicated to separating and/or reducing the non-productive labor involving software deployment that the users have to carry out in order to use the software. As software deployment gains more visibility, we foresee that more effort will be devoted to reducing the cost associated with it. Therefore, we designed the RADIUS application framework to make the software deployment process totally transparent to the users and to impose minimum overhead to the developers.

1.2. Design Principles

In the past few decades, a large amount of research effort in object-oriented systems was devoted to designing new object-oriented programming languages or new operating systems. Some high-caliber object systems such as COM [6, 15, 26, 41] or CORBA [29, 30, 31] offer very powerful reuse mechanisms, but the overhead involved in starting to use them cannot be justified considering the short lifecycle of many applications. For simple applications, these tools seem too complicated for programmers to overcome the initial learning curve. Powerful object-oriented programming tools are often not quickly accepted because the dominance of the market has very little to do with technical merit, but very much to do with compatibility. The users tend to pick the ones that are easy to learn and use, readily available and well supported. A tool can not gain wide acceptance quickly if it requires massive extensions to the operating system or the programming language.

Java [11, 14] has been successful so far, but the language design is not the major reason for its acceptance. It is popular because of the close resemblance to C++, cross-platform virtual machine, abstract windows toolkit and the class delivery mechanism [21, 22, 50]. We claim that for an object-oriented tool to be practically useful, it is not necessary to create new languages or operating systems. Novel ways of utilizing existing technologies will be appreciated far more by the real-world users.

Thanks to the speedy progress in RAD tools, it is becoming cost-effective to write single-use software. For example, an experimental computer scientist can build small animated presentations to demonstrate a new algorithm to a special interest group; an economist can use special software to display a newly found niche market to investment bankers; a medical scientist facing an epidemic outbreak can construct a program to analyze the samples collected and communicate the results to the hospital or the Center for Disease Control. Just like cable television catering to small groups of audiences, we foresee that software will be rapidly developed for small groups of users and many applications will undergo frequent, if not continuous, upgrading and refinement. For this new crop of throwaway applications, the deployment overhead involved in using them has to be reduced to near zero for them to be economically justifiable. Not only do we need a system that is lightweight on the users' side, we also need it to be light on the programmers' side because these programmers are not professional software engineers who are used to handling software deployment.

On the other hand, even software that is designed to be of long-term use can have a relatively heavy load of deployment tasks. Modern software projects tend to have users from early on for testing purposes. Many of them also have frequent releases [39] for bug fixes and feature improvements. The total amount of deployment work involved in the project is the number of users multiplied by the number of releases multiplied by the amount of deployment work in each release. For large software projects, a reduction in the last factor can have a great impact on the development cost and schedule. The author has first-hand experience with a software project in which a custom-made automatic upgrade module saved the company about half million dollars over three years.

For the reasons mentioned above, we chose to implement RADIUS with mature technology and made no extensions to either the programming languages or the operating systems. As long as the target operating system and programming language (and its compiler) support dynamically linked program modules, the implementation of RADIUS is very straightforward.

RADIUS is orthogonal to programming languages and tools. The minimal requirement in our design of RADIUS gives the developers the freedom to choose the tools to use while implementing their applications in the RADIUS framework. Even the documents can be easily made binary-compatible among programming languages and operating systems. Our approach is also orthogonal to other object systems, such as CORBA or COM, and is easy to integrate with them without conflict. For example, we can add the standard CORBA core functionality and Internet Inter-ORB Protocol (IIOP) [31] to a RADIUS system to make it inter-operable with other CORBA-compliant object systems over the Internet; or add an ActiveX [15] wrapper to the RADIUS Object Browser to make it function from within the Microsoft Internet Explorer. On the other hand, the functionality of RADIUS can be modeled as an interface (COM, CORBA and Java all use the same term) of other object systems and added to them as an enhancement.

In summary, RADIUS has the following attributes:

1. The system is based on a component structure so that the size of applications can be minimized and reusability can be maximized. New types of documents can be easily formed by using existing components as building blocks.
2. The system is centered on documents rather than applications. Applications are just tools to process documents. The users operate on documents and the applications will be located, delivered, installed and launched automatically on demand. The users view the system from a document perspective, not an application perspective.
3. The documents carry information about their applications in a canonical form independent of the underlying system so that the applications can be located even if they are not yet installed. This is very important for 1) lay users who have no idea about what kind of files are handled by which applications, and 2) a rapidly changing computing environment in which new applications and file formats show up at high frequencies.
4. The human overhead involved in using applications is zero. Applications are delivered over the network or the Internet and installed automatically when they are needed. Applications are upgraded automatically when there are new versions available. The users perform absolutely no work specific for software deployment unless they choose to.
5. The overhead involved in developing applications is also very light so that developers are not deterred by a steep learning curve. The users need just a few files to start using RADIUS, and the developers need just a few more to start developing RADIUS applications. No extensions are made to either the programming languages or the operating systems. The developers are not limited by which development environments they can use. All the developers have to do is to inherit an interface in the application class. When dealing with object-oriented technologies, very few tasks can be simpler than this.

1.3. Extending the Object-Oriented Programming Paradigm

The conventional way of using an application is to install the application first, then start creating, viewing and editing its documents. The code is directing how the data should be arranged. The object-oriented programming (OOP) paradigm suggests that this order should be reversed: the data should lead to the code. An object has a list of operations associated with the object. The execution is initiated by the objects, not the operations. However, almost all the software we see fails to meet this model of design at the point of invoking applications. While modern software often use OOP at the program level, the underlying operating system does not have the necessary mechanisms to support OOP at the document level, and therefore is not capable of locating the application for the document.

The basic entities in an OOP system are classes and objects. A class is a collection of declarations of data (a.k.a. fields) and procedures (a.k.a. methods). An object is an instance of a class. From an abstract point of view, an object can be thought of as a piece of data with some indications on what operations can be performed on the data. In most OOP language designs and implementations, these indications are pointers to methods. A pointer to an array of pointers to methods (virtual table pointer) is associated with the object in order to locate the methods for dynamic dispatching at runtime. In RADIUS, we extend this idea to the document level and introduce the idea of symbolic "pointers to applications" which indicate the location of executable application files over the Internet. The symbolic application pointers enable RADIUS to locate and gather the applications necessary for the documents without the need of installing the applications first, and thereby make software deployment totally transparent to both the developers and the users.

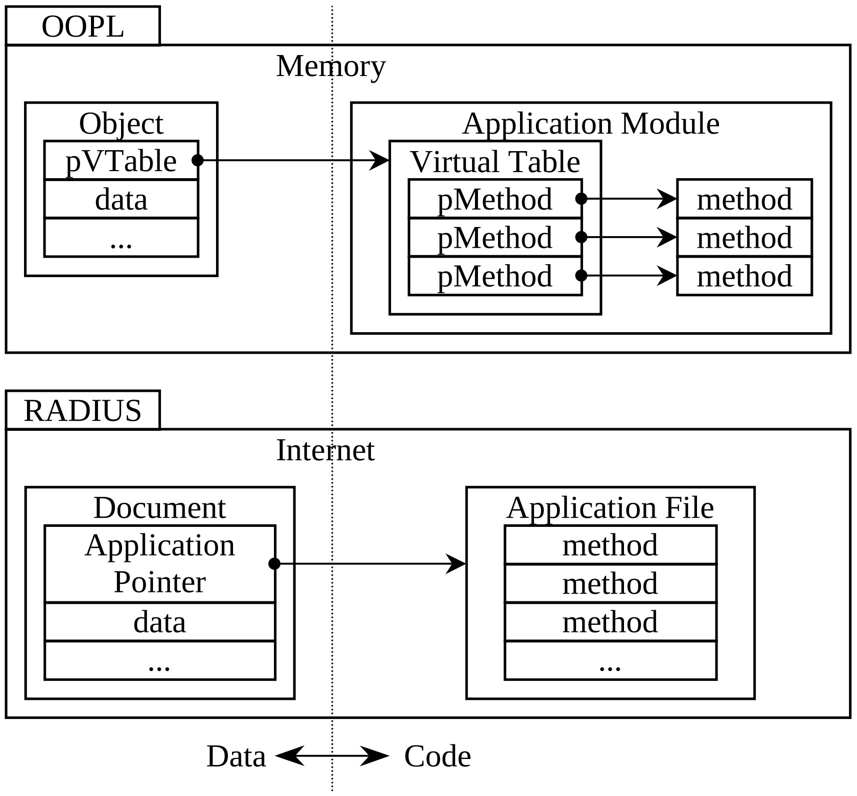


Figure 2: Analogy between RADIUS and OOPL

We designed the RADIUS application framework as means of automating software deployment tasks. The key technique in RADIUS is the application pointer.

Due to the heterogeneous nature of modern computing environment, we can safely assume that no hardware-specific or operating system-specific mechanism would be universal enough to solve the problem once and for all. As a result, we chose to use Uniform Resource Locators (URLs) to encode application pointers. Each document has an application pointer to indicate which application should be used to handle the document. In practice, an application module may be copied and stored in many places for various reasons. Therefore, we broke the application pointer down into two parts: the *Class Identifier* (the name of the class for the data stored in the document) indicating the type of the document and the *Application Locator* (a list of URLs paths, without the file name part) specifying on which servers the application module can be found.

When a document is processed (viewed, edited, etc.), RADIUS uses the Class Identifier and the Application Locator to find the application, create an appropriate object from the data contained in the document, and pass the object to the application. If the application is not found on the local client machine, the "application delivery and installation" process takes place through an interaction between the client machine using the document and the server machine specified in the Application Locator. If the application is found locally, RADIUS initiates the "application upgrade" process instead of an initial installation.

The machines on which the documents are used are called the RADIUS *Clients* and the machines specified in the Application Locators are called the RADIUS *Application Servers*. Through the RADIUS software residing on these two ends, the software deployment problem becomes transparent to developers and users.

1.4. Related Work

There has been some work done on issues related to software deployment [16]. However, none have tackled all the issues at once and none have given an integrated solution.

1.4.1. Installation Managers

There are quite a few software packages designed to facilitate the installation of software. InstallShield [10] and PC-Install [1] are the most famous. They provide easy ways for developers to write installation procedures, but the earlier versions did not address other aspects of software deployment. Later derivatives such as InstallFromTheWeb did address the delivery and upgrade issues, but still have not addressed the location issue.

Linux RPM [3] is a utility for specifying module dependencies of application packages and performing installation while maintaining the integrity of system setup. It doe not address the other issues.

The major deficiency of the installation managers is that the users still need to initiate the deployment operations explicitly. Moreover, they are tightly bound to the application files rather than the documents/data that need the applications.

1.4.2. Netscape Plug-in Finder and SmartUpdate

Netscape's Plug-in Finder is a simple search tool and database, which allows the users to look for plug-in modules that can be used to handle the downloaded content. It leads the users to the Web site from which the requested plug-in can be downloaded, but the users have to perform the download (delivery) and installation manually according to different instructions that come with each plug-in module.

SmartUpdate [27, 28] is a newer technique that allows the developers to package their applications and provide an “installation trigger” in the form of a piece of JavaScript contained in a Web page. The new Netscape Communicator browser has been designed to allow SmartUpdate packages to bypass Java Applet security checks in a limited way.

Oil Change™ [25] is a subscription service that allows the subscribers to keep their software up to date. It is a labor-intensive undertaking to ease the users’ burden on the upgrade tasks and is not automated on the developers’ end.

NSBD [9] (Not-So-Bad Distribution) is an automated Web-based distribution system that is designed for distributing free software on the internet, where users cannot trust the network and cannot entirely trust the maintainers of software. NSBD authenticates packages with “Pretty Good™ Privacy” (PGP™) or GNU Privacy Guard (GPG) digital signatures so users can be assured that packages have not been tampered with. NSBD’s focus is on security, leaving as much control as is practical in the users’ hands. It requires the developers to set up distribution packages manually and the users to run an update process manually.

1.4.3. Marimba Castanet

Marimba’s Castanet [24] products allow developers to build applications which are “broadcast” to whoever “tune” into corresponding application “channels.” It provides a convenient way for application delivery, installation and upgrade. However, a separate channel needs to be set up for each application and there is no well-defined relation between documents and applications. Several other systems [16] have been designed to work in similar ways.

1.4.4. Java Applications and Applets

Java [11, 14] applications are built into class files that are loaded on demand. The demands are generated by code, not data. The user specifies which application to run, then he/she can start using the application to process documents. Through the Serialization Interface [47] an object can be created from a data stream, but the system requires the class code to have been installed locally in order to de-serialize the corresponding object. There is no standard way of locating code from data before the class code is installed.

The Java applet framework [11, 21] is based on the idea of always delivering the latest version of the software and thus eliminating the need to install and upgrade applications. However, applets still cannot be located from documents. Furthermore, without a canonical caching scheme, the performance penalty involved in reloading the class files over a low-bandwidth connection is high. Applets operate in the same way applications do and with more security restrictions, which makes it hard to use applet classes from different sources in the same context.

Nevertheless, the Java class loading mechanism [21] showed us a way of building a similar technique to make Java applications automatically delivered, installed and upgraded.

Pal [33] describes an application delivery and upgrade mechanism for Java applications. It extends the Java class-loading mechanism to enable Java applications to be loaded in a manner similar to Java applets, yet caching the application files to avoid the performance penalty associated with reloading. It uses a “drop” mechanism that is in some way similar to the channels in Castanet. It is different from our approach in several aspects:

- It works with Java applications only. Our approach works with any object-oriented programming language that produces dynamically linked program modules.
- It needs a special purpose server (one per application) and associated programs to encode the upgrade information. Our approach uses a simple HTTP server.
- It requires a customized bootstrap program for each application, or the user has to input a server designator for each application. Our approach uses a unified object browser for all applications and the application information is encoded in the documents so that the users need not memorize the information.

1.4.5. Operating System Facilities

While none of the systems mentioned above provides a clear mapping between documents and applications, many operating systems do. Microsoft Windows and some Unix shells use a part (usually the suffix) of a document's file name to "associate" it with an application. This limits the freedom in naming document files. The MacIntosh operating system stores the application information with the document and thus does not post such a restriction on file names. These mappings work at the file level, not the object level, and thus do not work for compound documents that are assembled from other documents. The responsibility of digesting compound documents and locating applications for them are usually left to the applications or the component systems used to build the applications.

1.4.6. Component Systems

Component systems have become a major trend in software development. COM [6, 15, 26, 41], CORBA [29, 30, 31] and JavaBean [45] are the best-known ones in use. These systems provide a common foundation on which the developers can build interoperable and reusable software artifacts. While the developers can reuse existing components, the users rely on the developers' careful work to make available the components that are being reused. Software deployment is not a part of their standard services.

COM uses a centralized database (the Windows Registry) to store information about the installed servers for components [6]. If the server code of an object has not been installed, the object simply can not be created. The same goes for CORBA systems that rely on Implementation Repositories [29] to locate the object implementations. The Management Interfaces [30] specify how installation services should be organized, but the details about how installation services should be performed are left to the developers. It is noteworthy that this does leave the possibility for the integration of RADIUS into CORBA. The JavaBean [45] environment was designed to be a portable guest system running inside other native component systems. It also requires that the class files of the beans be installed (in the `CLASSPATH`) before use. The drawback of these application-centric designs is that they require applications to be installed before being used.

OpenDoc [2] is a document-centric component system design. It differs from the systems mentioned above by putting the focus on constructing documents or performing individual tasks, rather than using any particular application. The software that manipulates a document is hidden, and users feel that they are manipulating the parts of the document without having to launch or switch applications. The applications are considered "part handlers" associated with parts of documents, rather than independent identities. Nevertheless, OpenDoc still requires these part handlers be installed first and then be located through the Part Handler Registry.

None of these component systems provides a standard way of locating and delivering applications before they are installed. They all rely on a system-wide centralized database for component class registration and application location after the applications have been installed. If the registration database is damaged, the applications will need to be reinstalled even if the application files are already on the computer.

1.4.7. Miscellaneous Application-Specific Code

Automatic upgrade has been done in some custom software [49], including a sales-force automation system designed and implemented by the author. Some commercial software that also performs automatic upgrade has recently appeared on the market. Examples include Windows 98™ and Norton Utilities™. These upgrade facilities are proprietary code tailored for their applications and cannot be easily reused by application developers.

1.4.8. Summary

Compared against the existing systems, the most noteworthy attributes of RADIUS are:

- Works before the applications are installed.
- Guaranteed mapping from documents to applications.
- Zero-overhead on the user's end. No need to explicitly locate, install or upgrade applications.
- Minimal overhead on the developers' end. No need to explicitly construct installation packages.

The following table summarizes the features of RADIUS and other systems mentioned in this section.

	Comp. Systems	Java	Install Managers	SmartUpdate, Castanet, etc.	OS Facilities	App. Specific Code	RADIUS
Mapping Docs	*	-	-	-	+	-	++
Locate Apps	*	*	-	-	*	-	++
Deliver Apps	-	+	+	++	-	+	++
Install Apps	-	+	++	++	-	++	++
Upgrade Apps	-	+	-	++	-	++	++
Component Structure	++	+	-	-	-	-	++
Ease of Use	+	+	++	+	+	++	++
Ease of Dev.	-	+	+	-	-	--	++

++: Strong suit.
+: To a certain degree.
*: Only after installation of applications.
-: Weak, irrelevant or not available.
--: Very weak.

1.5. Thesis Organization

This dissertation is organized as follows: Chapter 2 illustrates the structure of the

RADIUS framework and its interaction model on an abstract level. The implementation in C++ on 32-bit Windows environment and Java are presented in Chapters 3. Chapter 4 demonstrates a presentation system and programming environment PIP (Programming in Presentation) built in the RADIUS framework. Application implementations at both the component (RADIUS) level and the script (PIP) level are revealed. The simplicity of design makes developing these applications relatively effortless. Chapter 5 summarizes our findings and points out future research directions.

Chapter 2: The RADIUS Application Framework

RADIUS [18] is based on the idea of extending in-memory code pointers to persistent "application pointers" which can be used to locate application files over the Internet. Through the standard services provided by the RADIUS system utilities, applications can be located, delivered, installed and upgraded automatically. The same techniques can be implemented in a variety of programming languages and operating systems, making RADIUS applicable to many software development environments.

2.1. The RADIUS Data Structure and Algorithm

As in any object system, the basic entities in RADIUS are *objects*. A RADIUS *document* is the persistent representation of a RADIUS object in the form of a file. A RADIUS document contains a *Class Identifier* specifying the name of the class of the object, and an *Application Locator* indicating where the application for the object can be found. A RADIUS *application* is a program file(s) containing the code for one or more classes (but only one of which is directly associated with the document). The applications are published on *Application Servers*, which form the server end of the RADIUS system. The Application Servers are essentially just file servers with versioning. In the simplest implementation, they are standard HTTP servers configured to deliver program files. The visual interfaces for editing and viewing RADIUS documents are created by the applications through the help of the *Object Browser*, which forms the client end of the RADIUS system. The Object Browser can be thought of as a graphical command shell for the operating system. It is in charge of locating, installing and upgrading the applications, and managing the creation of objects for the applications.

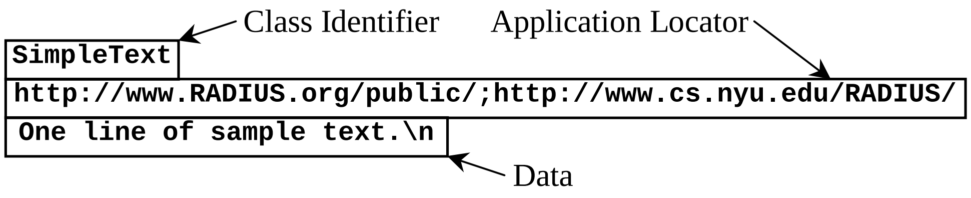

Class Identifier
Application Locator
SimpleText
http://www.RADIUS.org/public/;http://www.cs.nyu.edu/RADIUS/
One line of sample text.\n
Data

Figure 3: A Sample RADIUS Document

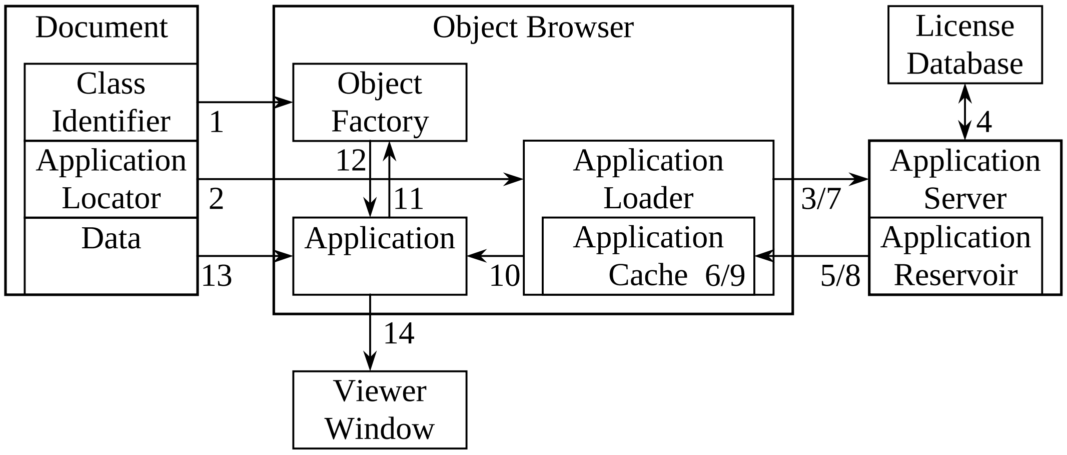


Figure 4: The RADIUS Process

To make an application RADIUS-compliant, a software developer should design the documents of the application to begin with a Class Identifier and an Application Locator, use the remaining part of the document for the application data, and implement the RADIUS methods in the application. After the application is implemented, it is published on an application server. Only some sample documents need to be distributed (maybe just posted on a Web site). When a user opens a document (either an original sample document or a modified document from another user) in the Object Browser, the Object Browser loads the application from the application server and installs (and later upgrades) it onto the client machine. This process is illustrated in Figure 4 and explained as the following steps:

1. The user asks the Object Browser to open a document. The Object Factory reads the Class Identifier in the document and tries to create an object of the specified class. If the Object Factory recognizes the Class Identifier, then the application is already in memory: go to step 12.
2. The Application Loader reads the Application Locator from the Document and looks for the Application in the Application Cache. If the Application is found, go to step 7.
3. The Application Loader sends a "request for application" message to the Application Server specified in the Application Locator.
4. The Application Server may optionally consult a software license database to ensure that the client machine is entitled to receiving the application and/or the upgrade.
5. The Application Server delivers the application file.
6. The Application Loader installs the application into the Application Cache and invokes any special installation procedure that the application may have. Go to step 10.
7. The Application Loader sends an asynchronous "request for upgrade" message to the Application Server while the process proceeds to step 10.
8. If the Application Server eventually returns an upgrade version of the application, the Application Loader puts it in temporary storage.
9. The Application Loader waits for an appropriate time to install the upgrade and invokes any special upgrade procedure contained in the application.
10. The Application Loader loads the application into memory.
11. The application registers itself with the Object Factory.
12. The Object Factory creates an object (with the help of the application) and passes it to the application.
13. The data portion of the document is read by the application.
14. A viewer window of the document is created by the application and the user can start viewing the document. The application decides whether the document can be edited and how the document is edited.

If an object contains other objects as part of its data, the process is applied recursively. An application is unloaded from memory by the garbage collector after all objects handled by the application have been destroyed.

An Application Locator may specify more than one Application Server, in which case each Application Server will be queried in turn until one sends a response to the client. For this reason, the Application Locator is sometimes called the "Application Path" in our implementations. The two terms are treated as the same.

```
static void install();
static void upgrade();
boolean view();
boolean readFromStream(InputStream stream);
static Object createFromStream(InputStream stream);
```

Figure 5: Key RADIUS Methods (in Java Notation)

The methods listed in the above figure are the key to achieving the RADIUS functionality:

1. If an application implements an `install` method, the method will be invoked at step 6 to perform application-specific installation tasks.
2. If an application implements an `upgrade` method, the method will be invoked at step 9 to perform version-specific upgrade tasks.
3. The `view` method is the entry point to the application. It does not need to literally display the document. If implemented, it will be invoked in step 14 to start the application. If an application does not implement this method, it will not be launched automatically by the Object Browser. The "document" in this case can be considered just a "trigger" to the deployment process.
4. The `readFromStream` method is used in step 13 to pass the data to the application. If an application does not implement this method, the Object Browser will not pass data to the application. However, the application will still be launched if it implements the `view` method. The "document" in this case can be considered a "shortcut" to the application.
5. If the `createFromStream` static method is implemented in the application and it returns a valid object upon reading from the data stream, the Object Factory will use it in place of the combination of creating an object and invoking `readFromStream` on the object. This combines steps 12 and 13 in the RADIUS process. The `createFromStream` method may be implemented straightforward as an object creation and a call to `readFromStream`, although that will be redundant.

In the most typical cases, the `view` method and one of the `readFromStream` and `createFromStream` methods should be implemented, while the `install` and `upgrade` methods are optional. To summarize, the following table shows all possible combinations:

`view` method	`CreateFromStream` or `readFromStream` method	Role of the document
Yes	Yes	A regular RADIUS document.
Yes	No	An application launcher only.
No	Yes	Not applicable. The class represents objects that should not be in a document alone. For example, a spell checker or a timer. The data should always be embedded in a parent object instead of being a document by itself.
No	No	An install/upgrade trigger only.

Figure 6: Roles of RADIUS Documents

```
String getClassID();
String getAppPath();
void setAppPath(String apppath);
String getFileName();
void setFileName(String filename);
boolean writeToStream(OutputStream stream);
boolean saveToFile(String filename);
```

Figure 7: Additional RADIUS Methods (in Java Notation)

The methods listed in the above figure are optional, but very useful, to the RADIUS functionality:

1. The `getClassID`, `getAppPath` and `setAppPath` methods are used to access the in-memory version of Class Identifier and Application Locator. Note that there is no method to change the Class Identifier, which is a string representation of the name of the class and thus should be constant. For programming systems that provide enough run-time type information, `getClassID` can be implemented using proper run-time facilities, such as `getClass().getName()` in Java or `typeid(*this).name()` in C++. Otherwise, the programmer should implement it in each class to return a constant string.
2. The `getFileName` and `setFileName` methods are used to access the filename associated with the document.
3. The `writeToStream` method is used to write the object to an existing stream. When the object is contained in another object, the parent object is responsible for creating (or inheriting) the stream. When an application streams out top-level objects as documents, it should use the `saveToFile` method. For read-only applications like Acrobat Reader™, these methods do not need to be implemented.

For the convenience of coding we group the RADIUS methods (Figure 5 and 7) into an interface in our implementations, but they can be accessed through other means (e.g. Java reflection or Windows DLL export) to make the application development more flexible.

2.2. The Application Server

A RADIUS Application Server is a file server over the Internet, with the added functionality of versioning and, optionally, software licensing and authentication. In our current design, an HTTP server configured to deliver DLL (for C++) and Java class files satisfies our basic needs. For commercial software, some licensing and authentication mechanisms need to be added. Since RADIUS keeps querying the Application Servers for upgrades after the applications have been installed, this even provides a way of catching software piracy.

Software licensing may not be an issue in some commercial situations, and we believe that such circumstances are likely to undergo significant growth. In the past few decades, computer software has grown to be more expensive than hardware. In many cases, the hardware can be considered "come with the software." Following this evolution, we can see another shift of value from the software to the information contained in or delivered by the software. One example is PDF – the reader software is free and the value is in the documents. We make a bold prediction that in a few more decades a considerable portion of software will be considered "come with the information." RADIUS is a perfect architecture for software in this paradigm. Of course, some software will never be given away, just like hardware today.

Another useful area for RADIUS where software licensing and code authentication may not be an issue is for Intranet applications, since these proprietary applications run on (virtually) private networks.

2.3. Storage Space Management

Modern commercial software has the tendency to take up a lot of disk space. One of the primary reasons for this "bloating" phenomenon is that the installation process is often treated as a one-time task. These applications have a large amount of code for many complicated functions and the installation processes take up considerable time and effort. The interdependency among modules often makes it necessary for the users to install everything at once, to avoid having to go through the installation process repeatedly. In RADIUS, the users spend absolutely no time and effort on installation. The object-oriented design of RADIUS breaks up code into many small components, which are installed only when used. In other words, no storage space is wasted on unused features.

In RADIUS, the applications are installed on a client machine by copying the application files into a cache directory. A bare bones RADIUS system does not need any kind of cache management. It just keeps installing applications until the storage is exhausted. For the reason mentioned in the previous paragraph, this may very well be all it needs. Nevertheless, some cache management schemes can be implemented for machines whose storage space is at a premium.

The extreme case of storage shortage is no storage. For diskless workstations, the cache directory can be set to a shared directory on the LAN, or be totally eliminated (i.e. no caching at all: program files are always loaded from the Application Servers) if network traffic and delay is not a problem for the target environment. Intranet applications whose Application Servers are on the same LAN as the client machines fall into this situation.

For machines that do have secondary storage, an alias table can be established to map application files to locations other than the cache directory. If the source of the application file is a readily-accessible secondary storage (e.g. local hard drive, non-removing CD-ROM, or LAN file server), then the Application Loader can simply add an entry in the alias table without actually copying the application file into the cache directory. Although many operating systems already have file alias mechanisms, they usually use an entire file to maintain an alias. They waste a lot of space by doing so. For example, a 64 byte one-line alias entry may take up 4096 bytes in the FAT32 file system of Windows 95, or 8192 bytes in a Unix file system. An alias table uses only about one percent of the space.

The Application Loader can also be programmed to limit the size of the cache space-wise or time-wise. It can remove application files that have not been used for a long time on a periodical basis or when the size of the cache reaches a certain quota. Aside from better managing the storage space, these implementations would also encourage "trying out" new applications since the users do not need to worry about space wasted on storing unused applications.

2.4. Wrapping Existing Applications into RADIUS

Wrapping existing applications into RADIUS is very easy. The programmer can write a “wrapper” class (which implements the RADIUS methods) to do the following:

- Use two String variables as the Class Identifier and the Application Locator. Implement methods to access them.
- A document is formed by adding the Class Identifier and the Application Locator in front of the real data file. Implement the `readFromStream` method to detach the rest of the stream into a real data file.
- Implement the `install` method to invoke the existing installation procedure of the existing application.
- Implement the `view` method to load/install the real application and start it with the real data file.
- When a document is saved, let the real application save the real document first, then use a `writeToStream` method to add the Class Identifier and the Application Locator in front of the document.

In most cases, the programmers don’t even need access to the source code of the real application to make this procedure work.

2.5. Summary

We presented the design of an object-oriented application framework that takes care of the software deployment problem. RADIUS has the following attributes:

- The identities initiating operations are the documents, not the applications.
- The users do absolutely nothing to install or upgrade the applications.
- Applications are installed only when needed.
- Applications can be delivered by existing HTTP servers.
- The use of object-oriented programming is extended to the file level.
- Application developers are not limited in what development tools to use.
- The dynamic design of the Object Factory eliminates the need for a centralized database (e.g. the Windows Registry).
- Extremely thin client. As shown in the following chapters, the C++ version has less than 700 lines of source code and 70 KB of executable binary (plus standard library routines). The Java version has less than 600 lines of source code and 25 KB of class binary (plus the standard Java classes).

Since RADIUS does not require any extension to programming languages and operating systems, it is very easy to integrate it with other software. The C++ version of RADIUS Developer’s Kit includes header files totaling less than 100 lines of code, and a 30-KB library file. The Java version of RADIUS Developer’s Kit contains only 16 KB of class binary.

While RADIUS is an application framework independent from the programming languages and operating systems, we will use some concrete examples to better demonstrate the underlying techniques. In the next chapter, we illustrate the implementation of RADIUS on two of the most popular platforms – C++ on 32-bit Windows, and Java. While these two languages are usually described as “similar”, we found that their implementations of RADIUS are quite distinct due to the differences in the underlying operating systems. The standard Java libraries provide many features to simplify some dynamic object-oriented tasks. This distinction confirms a general consensus that system facilities are usually more important than language features for developers. By making RADIUS a utility independent from programming languages and operating systems, we believe that it stands a better chance of being accepted by the user community.

Chapter 3: RADIUS Implementations

The RADIUS framework is independent from programming languages and operating systems. Any object-oriented programming system that supports dynamically linked program modules can be used to build a RADIUS environment. In this chapter, we describe two implementations on the most popular platforms – C++ on 32-bit Windows, and Java.

For each running instance of a RADIUS implementation, exactly one instance of the Object Browser is created. Within an Object Browser, the Object Factory and the Application Loader are both singletons [12]. Therefore, the implementations can choose to program the Object Factory and Application Loader methods as global functions (but not in Java, of course), class methods of the Object Browser or instance methods of the Object Browser. The choices shown in this chapter were made just for the convenience of our coding.

3.1. The RADIUS Programming Interface

The RADIUS Programming Interface contains the methods needed to achieve RADIUS functionality. These methods have been briefly explained in the previous chapter. Here we discuss the issues specific to each implementation in more detail.

3.1.1. The RADIUS Programming Interface in Java

RADIUS applications written in Java are compiled into Java class files. Following the design philosophy of Java, RADIUS classes in Java do not, nor should they, derive from one common base class. A RADIUS class in Java is any class that implements the RADIUS interface, whose definition is given in the following figure.

```
public interface RADIUS {
  String getClassID();
  String getAppPath();
  void setAppPath(String apppath);
  String getFileName();
  void setFileName(String filename);
  boolean readFromStream(InputStream stream);
  boolean writeToStream(OutputStream stream);
  boolean saveToFile();
  boolean view();
}
```

Figure 8: RADIUS Interface in Java

While this design prohibits default implementation of the methods, it does give the programmers greater freedom in designing their applications. Moreover, the standard implementation of methods is short enough, as can be seen later, to be textually copied without difficulty.

Although the Java object serialization interface [47] provides an easy way of streaming objects, the methods involved use a file format that is specific to Java. If developers want their Java RADIUS documents to be binary compatible with RADIUS applications implemented in other languages, they need to implement their own streaming routines in the Java RADIUS applications, or have applications implemented in other languages read and write data according to the Java Serialization format. If binary compatibility of documents is not an issue, the developers may simply define the application class to implement the `java.io.Serializable` interface and invoke the `readObject` and `writeObject` methods from within the `createFromStream` (not `readFromStream`) and `writeToStream` methods.

The figure below illustrates the standard implementation of RADIUS applications in Java. The programmers can copy this template code for any new RADIUS application and just implement the application entry point. If the application is not required to derive from some other class (e.g. a visual component), then the programmer can even derive the application class from this one.

```
// standard implementation
class RADIUSObject implements RADIUS, java.io.Serializable {
  protected transient String AppPath, FileName;
  public String getClassID() { return getClass().getName(); }
  public String getAppPath() { return AppPath; }
  public void setAppPath(String apppath) { AppPath=apppath; }
  public String getFileName() { return FileName; }
  public void setFileName(String filename) { FileName=filename; }
  public static Object createFromStream(InputStream stream) {
    return new ObjectInputStream(stream).readObject();
  }
  public boolean readFromStream(InputStream stream) {
    // may not be necessary if createFromStream is implemented
    return false;
  }
  public boolean writeToStream(OutputStream stream) {
    DataOutputStream dos=new DataOutputStream(stream);
    dos.writeUTF(getClassID());
    dos.writeUTF(getAppPath());
    new ObjectOutputStream(stream).writeObject(this);
  }
  public boolean saveToFile() {
    FileOutputStream fos=new FileOutputStream(FileName);
    boolean b=writeToStream(fos);
    fos.close();
    return b;
  }

  // optional
  public static void install() {
    // customized installation tasks
  }
  public static void upgrade() {
    // customized upgrade tasks
  }

  // the programmers just implement the application entry point
  public boolean view() { /*...*/ }
}
```

Figure 9: Standard RADIUS Application in Java

3.1.2. The RADIUS Programming Interface in C++

RADIUS applications written in C++ on Win32 are compiled into Dynamically

Linked Libraries (DLLs). The application classes are derived from the base class `RADIUSObject`, whose definition is listed in the following figure. The methods have been given default implementations so that the programmers can just omit the methods that they do not want to implement.

```
class RADIUSObject
{
protected:
  String AppPath, FileName;
public:
  virtual String getClassID(void) { return typeid(*this).name(); }
  virtual String getAppPath(void) { return AppPath; }
  virtual void setAppPath(String apppath) { AppPath=apppath; }
  virtual String getFileName(void) { return FileName; }
  virtual void setFileName(String filename) { FileName=filename; }
  virtual bool readFromStream(TStream *stream) { return true; }
  virtual bool writeToStream(TStream *stream) { return true; }
  virtual bool saveToFile(void) { return true; }
  virtual bool view(void) { return false; }
};
```

Figure 10: Definition of the RADIUS Base Class in C++

In addition to inheriting from the base class `RADIUSObject`, RADIUS applications in C++ need to include some standard supporting code, which are listed in Figure 11. Most important of all, a virtual constructor [12] must be provided in order for the objects to be created from outside the application without static type information. The Object Factory needs to create dynamically arbitrary type of objects without having access to the declaration of the class. Since the standard C++ run-time type system does not provide a reflection facility [46], a developer-defined virtual constructor is necessary. This virtual constructor is passed to the Object Factory as part of the application registration (step 11 of the RADIUS process) and is used later by the Object Factory to create objects for the application.

The `createFromStream` method is optionally implemented as a DLL export function, since functions cannot be static and virtual at the same time. This “streaming virtual constructor” can be conveniently used when the object does not have a default constructor (one that has no parameters).

The `install` and `upgrade` methods are also implemented as DLL export functions. If the Application Loader detects their existence, they will be invoked at appropriate times: `install` will be invoked after the application module is copied into the client machine for the first time, and `upgrade` will be invoked after the application module replaces an older version on the client machine.

Another optional part of the application is an object-count mechanism used to support application unloading, which has to be implemented explicitly, since standard C++ (and the underlying Win32 operating system) does not provide a garbage collector to do so. This classic object-count technique is described in most C++ textbooks [43]. The actual unloading mechanism is explained later in the Application Loader. Without this part, the application file will remain in memory until the Object Browser terminates, which may in fact be acceptable in some cases.

```
class MyObject : public RADIUSObject
{
  // the application class
};

// virtual constructor
extern "C" __declspec(dllexport) void *VirtualConstructor(void);
void *VirtualConstructor(void) { return new MyObject; }

// optional: createFromStream method
extern "C" __declspec(dllexport) void *createFromStream(Stream *s);
void *createFromStream(Stream *s)
{
  // create an object from the stream ...
}

// optional: install and upgrade methods
extern "C" void install(void);
void __export install(void)
{
  // customized installation tasks
}
extern "C" void upgrade(void);
void __export upgrade(void)
{
  // customized upgrade tasks
}

// optional: object count mechanism for application unloading
long MyObject::ObjectCount=0; // class variable
MyObject::MyObject()
{
  ObjectCount++;
  // continue with object construction ...
}
MyObject::~MyObject()
{
  // object destruction ...
  if (--ObjectCount==0)
    unloadApplication(getClassID());
    // defined in Application Loader
}
```

Figure 11: Sample RADIUS Application in C++

3.2. The Application Loader

The Application Loader is in charge of loading the application files into memory on demand, and actually locating, receiving, installing and upgrading the application files. It maintains a list of applications that have been loaded into memory. This list is consulted implicitly by the Object Factory when an object needs to be created. If the application handling the object class is not on the list, the Application Loader loads the application file into memory.

The "installed" application files are stored in a cache directory on the client machine. When the Application Loader loads an application file, the cache directory is searched for the application file. If the application file is found, it is assumed that the application has been installed and the “upgrade” process takes place; otherwise, the “locate and install” process takes place. In either case, the application file is loaded into memory as soon as it becomes available.

During the “locate and install” process, storage locations contained in a list (search path) are searched first before the Application Loader goes out to the Internet site(s) specified in the Application Locator. This search path allows the users or system administrators to redirect network traffic to “proxy” Application Servers for many possible reasons:

- Install from CD/DVD or other removable storage media, for applications distributed in the conventional way. Considering the amount of storage on these media, they actually have a huge bandwidth that should not be overlooked.
- Install from LAN file servers or caching proxy servers, for centralized configuration control or reducing network traffic. The system administrator may want to make sure the upgrades are safe before the users are allowed to access them, or want to make sure that all the users are using the same versions.

The methods in the Application Loader can also be invoked explicitly to install and upgrade non-RADIUS applications. This mechanism allows programmers to use program modules developed in the conventional way while enjoying the convenience provided by RADIUS. The only difference is that non-RADIUS classes do not participate in the RADIUS process beyond the deployment portion.

3.2.1. The Application Loader in Java

```
public class Loader extends java.lang.ClassLoader {
  public Class loadClass(String name, boolean resolve)
    throws ClassNotFoundException;
    // required by java.lang.ClassLoader
  public Class loadApplication(String classid, String apppath);
  public Class loadApplication(String appname);
  public String getCacheDir() { return CacheDir; }
  public void setCacheDir(String dir) { CacheDir=dir; }
  public String getSearchPath() { return SearchPath; }
  public void setSearchPath(String path) { SearchPath=path; }
}
```

Figure 12: Key Portion of Application Loader in Java

The `loadClass` method is required by the Java Class Loader mechanism. It takes the name of the class as the first parameter. The second parameter indicates whether to link the class.

The first `loadApplication` method saves the Application Locator (passed in by the `apppath` argument) in a private variable and then calls `loadClass`, which then reads the Application Locator to decide where to load the class file from. If the `loadClass` method is invoked recursively at this point due to class code resolution, the same Application Locator will be reused and load the classes from the same host.

The second `loadApplication` method is used internally for loading applications that are known to have been installed. The remaining methods are involved with the application cache directory and the search path.

Both forms of the `loadApplication` methods returns a `Class` object which can be used by the Object Factory to create objects of the application class.

At a first glance, it seems like a simple task to extend the Java `ClassLoader` class to build the RADIUS Application Loader. After some investigation and experimentation, we realized that the design problem is in fact quite subtle. To explain the difficulties, we first identify the characteristics of the Java Class Loader mechanism [21, 22, 50] that affected our design of the Application Loader:

A. If a class file exists in the local `CLASSPATH`, the Java Virtual Machine treats it as a “system class” and does not use a `ClassLoader` object to load it.

B. If class X uses class Y and class Y has not been loaded, the Java Virtual Machine will ask the `ClassLoader` object which loaded class X to load class Y with the `loadClass` method. A class derived from the `ClassLoader` class must implement the `loadClass` method, which takes just two parameters – a `String` identifying the name of the class, and a `boolean` flag for whether to resolve (link) the class. This method is implicitly called by the Java Virtual Machine when a class file needs to be loaded, and it must then decide where to find the class file.

C. A class must be initialized at the time of its first active use[*]. The sequence of events is: load, link and initialize. Suppose class X imports class Y. Then class Y can be loaded and linked as early as class X is being loaded or as late as class X is initialized. However, if class X extends class Y, then class Y will be loaded when class X is loaded, otherwise class X cannot be verified.

D. Although the underlying Java Virtual Machine may choose to perform loading and linking early (as early as possible) or late (as late as possible), it must give the impression that it is performing them late. If an error is encountered during the early loading or linking of a class, it must not be reported until the time as if the JVM were performing late loading or linking.

E. For each `ClassLoader` object and the classes loaded by it, the Java Virtual Machine creates a separate address space. Instances of the same class loaded by different `ClassLoader` objects are considered to be different internal types. The internal types are identified by the pair (*loader*, *class*).

This architecture works fine when the class loader knows exactly where to find the class files at all times. Some examples of this kind of situations are: 1) applet class loaders, which load an applet from a host machine and then load all subsequent classes from the same host machine; 2) RMI (Remote Method Invocation) class loaders [48], which load local “stub” classes that themselves know exactly which remote hosts to talk to. However, we found this design insufficient for the RADIUS model, which has a more dynamic paradigm for the locations of applications.

A Java application is usually installed into a location on the `CLASSPATH`. For RADIUS applications, this does not work because of (A). Since there will be no class loaders for these “installed” applications, we will not have a chance to intercept the class loading in order to perform the automatic upgrade. Therefore we store the class files in a cache directory separate from the Java `CLASSPATH` and let our class loader – the

[*] Active use of a class is defined as the 1) invocation of a constructor of the class, 2) creation of an array of the class, 3) invocation of a method declared (not inherited) by the class, or 4) use or assignment of field (except static final fields) declared by the class. Active use of an interface is defined as the use or assignment of field declared by the interface.

Application Loader – to load them on demand.

An application class usually uses other classes to help its tasks. In C++ all these classes can be compiled into one single application module and loaded all at once. Java classes are compiled into separate class files[*], therefore we designed the Application Loader to be a subclass of `java.lang.ClassLoader` so that we can take advantage of (B) in order to load the classes being used automatically. We wrapped the `loadApplication` method around the `loadClass` method. Although the `loadClass` method does not take a parameter as the Application Locator, we have the `loadApplication` method store the Application Locator in an instance variable and let the `loadClass` method refer to the variable when searching for the class file.

The flexibility specified in (C) became an obstacle to us. At the time when the JVM asks our Application Loader to load a class Y, we may not know which class X triggered this request and therefore we do not know which Application Locator to use. We first assume that class X and class Y came from the same host, then the problem can be divided into two situations:

1) Class X extends class Y. Class Y is loaded when class X is loaded. We then have a valid Application Locator stored by the `loadApplication` method, so class Y can be found automatically.
2) Class X does not extend class Y but has active uses of class Y. Class Y will be loaded between the time class X is loaded and the time class X is linked. (Although the specification requires the JVM to initialize class Y only when class X is initialized, JVM needs to resolve the links and determine if class X has the access permission to use class Y at the time class X is linked.) Fortunately, we can force the linking (and even initialization, although unnecessary for our purpose) of class X to take place at the time class X is loaded by invoking the `resolveClass` method inside the `loadClass` method (and therefore ignoring the second parameter). At this time (before the loading of class X finishes), we still have a valid Application Locator stored by the `loadApplication` method, so class Y can be found automatically. Note that the `resolveClass` method is invoked in the `loadClass` method, not in the `loadApplication` method, so this mechanism works recursively on classes used by class Y.

The solution for (2) does violate (D) if we were implementing a JVM because class loading and resolution errors will be reported by our Application Loader too early due to the fact that we always resolve any classes we load. Nonetheless, we are implementing a class loader, not a JVM.

This solution works when the referencing class and the referenced class reside on the same Application Server. Serious problems arise when the classes reside on different Application Servers. Suppose class X (served by host H1) references class Y (served by host H2) in method M. According to (D), we should be able to run the code in Figure 13 because by the time the constructor of class Y is invoked, the class file of Y should have been installed and therefore class Y should be able to be initialized without error. However, the designers of JVM did not take into account the possibility of a class

[*] We could put all related classes into a JAR (or ZIP) file, but that will prevent other classes from reusing the classes contained in this JAR file.

changing its validity between the time of early linking and actual initialization. In the code listed below, class X will be reported to have an error referencing to class Y because when class X is loaded, we have no way of knowing that class Y should be found on server H2 and therefore cannot even verify class X.

```
import Y;
public class X {
  public void M() {
  {
    // style-1
    loadApplication("Y","H2");
    Y y1=new Y();
    // or style-2
    Y y2=createObject("Y","H2");
    // ...
  }
}
```

Figure 13: Dependency among Classes from Different Hosts

We do have a solution to this problem, but we want to point out first that there is no technical reason for class Y and class X to be served by different servers! In order for class X to be compiled, the programmer has to have a copy of class Y anyway. If for any (most likely legal) reason the programmer cannot redistribute class Y, then the following (ugly) solutions can be used.

2') In class X declare a constant String field whose value is a special string indicating that class Y should be loaded from server H2. When the Application Loader loads class X, it first reads the binary data of class X to see whether such "dependency" strings exist; if so, the referenced classes are loaded before X is resolved.

2") Alternatively, the programmer can write a wrapper class that does not depend on either class X or class Y to install both classes.

Neither solution is clean and easy to use. The implementation of (2') also involves decoding the Java class files, which we would like to avoid. For the time being, we have implemented the rule that an Application Server must serve all the classes used in all the applications it serves. In the previous example, the Application Server (H1), which serves class X must also serve class Y. For a permanent solution, we propose the following extension to Java: extend the syntax of the `import` statement to support an optional Application Locator path. For example, instead of just:

```
import Y;
```

the programmer should be able to write:

```
import Y("http://www.RADIUS.edu/");
```

so that the extended Java Virtual Machine will know to look for `Y.class` at `http://www.RADIUS.edu/`. In addition to being used at runtime as the Application Locator, this feature can also be used at compile time to locate the latest version of the classes being imported. Of course, if the path is not specified, it is assumed that the classes reside on the same host.

The feature (E) was designed as a security mechanism. It sets up a sandbox for a class to operate within, and prevents the class from accessing classes in other address spaces. However, it forces us to use the reflection interface to assure type identity in some cases. When an application wants to load another class and interact with it, the application cannot use another Application Loader object to do so. It must use the Application Loader that loaded itself. Otherwise, the newly loaded class will be loaded into a separate address space to which the application has no access. However, when an application invokes the `loadApplication` method of the Application Loader that loaded itself with the code in the following figure, the type cast `(Loader)` will fail due to a type mismatch. Assume that the system Application Loader is L1. The internal type of L1 is (*`null,RADIUS.Loader`*). L1 loaded `MyApp`. The type cast asks L1 to load the class `RADIUS.Loader`, which yields internal type (*`L1,RADIUS.Loader`*). The `getClass().getClassLoader()` returns L1 of type (*`null,RADIUS.Loader`*) which cannot be cast to (*`L1,RADIUS.Loader`*).

```
import RADIUS.Loader;
public class MyApp {
  public void MyMethod() {
    ((Loader)getClass().getClassLoader()).loadApplication(...);
  }
}
```

Figure 14: Caveat in Using the Application Loader in Java

Without the type cast, we cannot explicitly invoke the `loadApplication` method of the Application Loader. With the type cast, we have a type conflict. The solution is to use the Java reflection interface to retrieve a `Method` object (for the `loadApplication` method) from the system Application Loader object to invoke the `invoke` method with the intended arguments. Fortunately, all these details are encapsulated in the Object Browser and the programmer will only see that there are methods (with identical signatures) available for use in the Object Browser, as listed in the following figure.

```
  public Browser() {
    SystemLoader = getClass().getClassLoader();
    LoaderClass = SystemLoader.getClass();
    String t="";
    Class[] StringClass2={t.getClass(),t.getClass()};
    LoadApplication2 = LoaderClass.getMethod(
      "loadApplication",StringClass2);
    // ... other initialization tasks
  }

  public static Class loadApplication(String classid,
    String apppath) {
    Object[] a={classid,apppath};
    try { return (Class)LoadApplication2.invoke(SystemLoader,a); }
    catch (Exception e) { return null; }
  }
```

Figure 15: Code in Object Browser to Encapsulate the Application Loader

The Java version of RADIUS starts with the Application Loader class, which creates an instance (and thus loads the class) of the Object Browser in its constructor. Since the Object Browser is loaded by the Application Loader, all other classes loaded by the Application Loader can communicate with the Object Browser without the aforementioned type mismatch.

3.2.2. The Application Loader in C++

Before explaining the Application Loader in C++, we must first introduce a meta-object construct we implemented for C++. The class `Class` is designed to be the bridge between the RADIUS run-time system and the virtual constructor (and optionally the streaming virtual constructor) in the application module. It mimics the Java class `java.lang.Class` in some way, but we only implemented the functions we need, and just did a little more for the "streaming virtual constructor," in order to implement the `createFromStream` method.

```
typedef RADIUSObject *(*VC)(void); // virtual constructor
typedef RADIUSObject *(*SVC)(Stream *); // streaming VC
class Class
{
private:
  String ClassID;
  VC Constructor;
  SVC StreamingConstructor;
  HINSTANCE Module;
public:
  Class(String classid, String apppath, VC vc, SVC svc,
    HINSTANCE module) : ClassID(classid), AppPath(apppath),
    Constructor(vc), StreamingConstructor(svc), Module(module) {}
  ~Class() { FreeLibrary(Module); }
  String getName(void) { return ClassID; }
  String getPath(void) { return AppPath; }
  HINSTANCE getModule(void) { return Module; }
  RADIUSObject *newInstance(void)
  {
    return Constructor==NULL?NULL:Constructor();
  }
  RADIUSObject *newInstanceFromStream(Stream *s)
  {
    return StreamingConstructor==NULL?NULL:StreamingConstructor(s);
  }
};
```

Figure 16: Simple Meta-Object `Class` in C++

The C++ version of RADIUS was first implemented in a different fashion [17]. It was a little more efficient, but more complicated. Later, when implementing the Java version, the absence of explicit method pointers forced us to resort to the meta-object feature. After the Java version was done, we felt compelled to rewrite the C++ version to use a similar mechanism, and ended up cutting the size of the code in half. If the C++ RTTI (Run Time Type Information) system were extended to include a virtual constructor (or links to real constructors), then there would be no need to re-implement this meta-object wrapper.

One `Class` object is created for each application. The `Constructor` member is "dynamically linked" to the exported function `VirtualConstructor` in the application DLL file by the Application Loader when the application module is first loaded, and the `StreamingConstructor` member is linked to the "streaming virtual constructor" – the `createFromStream` function. By using this meta-object wrapper, we were able to unify the signature of the C++ and Java implementations of RADIUS.

```
namespace ApplicationLoader
{
  Class *loadApplication(String classid, String apppath);
  Class *loadApplication(String appname);
  String getCacheDir(void);
  void setCacheDir(String dir);
  String getSearchPath(void);
  void setSearchPath(String path);
  void unloadApplication(String classid);
  void sweepApplication(void);
}
```

Figure 17: Definition of Application Loader in C++

The Application Loader maintains a list of loaded applications by storing the `Class` objects of the applications in a list. This list is looked up when the Object Factory needs to create objects.

The first `loadApplication` method is the core of the Application Loader. It returns a meta-object of class `Class` whose `newInstance` method can be used to create objects of the requested application. The second form of the `loadApplication` method is used when an application has already been installed, and the caller does not have a document containing an Application Locator. An example of this situation is when the application is launched in the conventional way – by launching the application file itself, rather than through a document. In this case, the application file must reside in the cache directory or on the search path.

The `loadApplication` methods return the existing `Class` objects for applications that have been loaded. If the requested application has not been loaded yet, they load the application, create and return a new `Class` object that is added to the internal list.

The next four methods (`getCacheDir`, `setCacheDir`, `getSearchPath` and `setSearchPath`) are just access methods to the cache directory and search path settings. The last two (`unloadApplication`, `sweepApplication`) are designed to compensate the absence of a garbage collector in C++. They do not exist in the Java version. Since the design philosophy of Java is not to let the programmers handle memory management, we let the Java garbage collection mechanism manage memory reclamation. Of course, RADIUS applications implemented in Java do not need to worry about unloading themselves.

Each application should maintain an object count (see Figure 11) to control the unloading of the application. When the last object handled by the application is destroyed (the object count reaches 0), the destructor of the object invokes `unloadApplication` to mark the application DLL file as "can be unloaded." On the next invocation of the garbage collector, the `sweepApplication` method will be called to actually unload the application. This extra step is necessary because a Windows DLL file cannot unload itself. If an unload call is made from within the DLL, the call will return to the code that has already been unloaded and cause an access violation. The `unloadApplication` method also severs the mapping from the class name to the `Class` meta-object by removing the application from the loaded list, thus preventing new objects of the application from being created. If new objects of the application are indeed created, they will be created by another instance of the application (may even be an upgraded one), and therefore will not hinder the removal of the application by `sweepApplication`.

The current implementation of RADIUS invokes the garbage collector periodically in the Object Browser through a timer event. In general, the only requirement is that the garbage collector be serialized with the constructors and destructors to avoid race conditions.

An application may choose not to implement an object count. In that case, the DLL file will simply remain in memory (physical or virtual) until the Object Browser is shut down, unless `unloadApplication` is invoked explicitly. However, if an object is still in use after the DLL has been removed, the program will crash when the code of the object is accessed. Therefore, explicit calls to `unloadApplication` must be handled with extreme care.

When an application uses objects of another class, the programmer must keep in mind that the code modules are dynamically linked. A few stylistic coding rules must be observed:

- We try to avoid using import libraries because they require extra steps when automatic software deployment is added to applications. Without import libraries, the objects of other classes should be accessed only through run-time pointers. To be more specific, only data members and virtual methods should be accessed. Access to non-virtual instance methods, static data members and static methods are prohibited.
- The `new` operator should not be used to create objects of other classes. All “foreign” object creation should be performed through the Object Factory.

These attributes are actually common to most component systems. Developers who are familiar with COM, CORBA, or other popular component environments should already be accustomed to these programming styles.

Let’s review the dependency problem we encountered in the previous subsection. The style-2 code in Figure 13 actually works because in C++ the class Y does not need to be loaded until an instance of Y is actually created, at which time the location information can be provided to the Object Factory. The lack of class code verification in C++ is a potential loophole in the type system, but it solves our problem. Style-1 code does not work because it uses the non-dynamic `new` operator, which requires static linking. However, we can rewrite it as the RADIUS style code:

```
Y *y1=loadApplication("Y","H2").newInstance();
```

which will work (the Java version can be rewritten the same way).

Sometimes programmers do want to use import libraries to reuse existing code. If an application uses a DLL through an import library and the DLL cannot be found locally, Windows will not allow the application to load. Since Windows does not provide a way like the Java Class Loader mechanism to indicate which application files are missing, the DLL cannot be found automatically. For this kind of situation, we devised a different mechanism to specify module dependency.

When an application fails to load due to the absence of DLL files, the application is loaded as a data module, which does not get executed and therefore does not require linking to the DLL files. The string table resource of the data module is searched for strings whose identifier numbers fall within a certain range reserved for dependency strings. These dependency strings are then interpreted as Class Identifiers and Application Locators and passed to the `loadApplication` method for the DLL files to be installed. The data module is then unloaded and the application loading is attempted again. This mechanism works recursively so that the referencing application will only need to specify the applications it references directly.

```
#define DEPENDENCY 17488 // arbitrary choice, =='DP' for dependency
STRINGTABLE
{
  DEPENDENCY, "PIP_Picture http://www.RADIUS.edu/PUBLIC/"
}
```

Figure 18: Sample Windows Resource Script Specifying DLL Dependency

As an example, the figure above shows a resource script that would be included in applications that use `PIP_Picture.DLL` (located at `http://www.RADIUS.edu/PUBLIC/`) through an import library (`PIP_Picture.lib`). The fact that `PIP_Picture.DLL` references another module `PIP_Object.DLL` does not require the applications to add a line specifying the dependency on `PIP_Object.DLL` because it has been taken care of by the `PIP_Picture` application. If more than one DLL is referenced by the application, more lines can be added using `DEPENDENCY+1`, `DEPENDENCY+2`, etc.

3.2.3. Naming Application Files

Application files are stored in the cache directory without the Application Path information. When the Application Loader looks for the application file in the Application Cache, only the class name of the application is used in the comparison. Therefore, the name of each application file should be made unique to avoid conflicts. Although many systems use OSF DCE style 128-bit Universal Unique Identifier [23] to uniquely identify classes and then provide mappings to human readable names, we felt compelled that the human readable names should be unique to begin with.

Sun proposed a naming convention for Java classes to prevent name conflicts. The convention states that the name of a class should begin with the Internet domain name of the developer of the class separated by dots in reverse order. For example, a Java class developed by the Computer Science Department of New York University should begin with `EDU.NYU.CS` (or `edu.nyu.cs`). When the class file is stored on disk, the dots are replaced by proper path separator character(s). Throughout this thesis, we just use short descriptive names for demonstration purpose, but for practical applications, we propose the use of a naming convention of C++ application files similar to that of Java's. Since the dot character is not a valid part of identifier names in C++, we can use the underscore character ('_') instead. Therefore, C++ classes developed by the Computer Science Department of New York University will begin with `EDU_NYU_CS` (or `edu_nyu_cs`). Our current implementation does not replace the underscores by path separator characters for coding convenience.

3.3. The Object Factory

Object factory is a pattern [12] commonly seen in object-oriented systems. The original idea was to provide a dynamic mechanism for creating objects whose exact classes are unknown at compile time but are known to be from a limited set. The factory will know exactly which class of object to create at run time. Therefore, it is commonplace to program the core of the factory as a selection statement with each creation task hard-coded. For example, Unidraw [51] uses a factory mechanism to reconstruct objects from data saved on disk. By storing a Class Identifier before the instance data, Unidraw is able to identify which class of object to create when the data is read back from disk, and therefore is able to create an object from a predefined domain-specific set of classes.

We extend this mechanism to implement an Object Factory that can create arbitrary classes of objects by dynamically loading the application modules for the object classes. The key to this functionality is the combination of the Object Factory and the Application Loader. By registering a reference to the constructor of a class in the Application Loader when the application is loaded, the Object Factory is capable of creating arbitrary class of objects without having access to type information at compile-time.

In our current implementations, the references to constructors are in the form of meta-objects, but straightforward function pointers can also be used if the underlying programming language supports them. That was actually the way we first designed the C++ version of RADIUS.

Another advantage of using an Object Factory is that instead of letting the applications create objects as they wish, anytime and anywhere, all object creation is handled in a centralized fashion through the Object Factory. Although we are currently not utilizing this centralized control feature, it is very useful if developers want to plug in object management services in the future.

3.3.1. The Object Factory in Java

```
public static RADIUS createObject(String classid, String apppath);
public static RADIUS createObject(String classid);
public static RADIUS createFromStream(InputStream stream);
public static RADIUS createFromFile(String filename);
```

Figure 19: Object Factory in Java

The first `createObject` method is the core of the Object Factory. It takes the Class Identifier and the Application Locator as arguments and pass them to the Application Loader to retrieve a `Class` object for the application, and then invokes the `newInstance` method of the `Class` object to create the requested object. The second `createObject` method is used when the application is known to have been installed, as in the case of the second form of the `loadApplication` method of the Application Loader. In essence, it is used in place of the `new` operator.

Creating objects of unknown classes becomes easier with the meta-object construct. The `newInstance` method of the class `Class` serves as a universal virtual constructor for all classes and thus eliminates the need for hard-coding virtual constructors.

The `createFromStream` method creates an object from a stream. It reads the Class Identifier and the Application Locator from the stream and retrieves the `Class` object of the application from the Application Loader. If a streaming virtual constructor is defined for the application, it is invoked to create the object directly from the stream, otherwise the virtual constructor of the application class is invoked to create an object and the `readFromStream` method of the object is invoked to read the data from the stream. The `createFromFile` method creates an object from a file by opening the file as a stream and then invoking `createFromStream` on the stream.

3.3.2. The Object Factory in C++

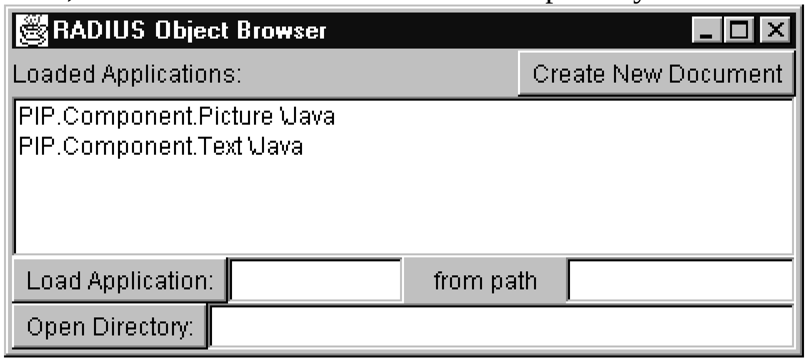
```
namespace ObjectFactory
{
  RADIUSObject *createObject(String classid, String apppath);
  RADIUSObject *createObject(String classid);
  RADIUSObject *createFromStream(InputStream *stream);
  RADIUSObject *createFromFile(String filename);
};
```

Figure 20: Definition of Object Factory in C++

The C++ version of the Object Factory has the same operations as the Java version, except that the methods are implemented as global functions instead of class methods of the Object Browser. Once the meta-object construct is implemented, subsequent coding is almost a line-by-line translation of the Java version.

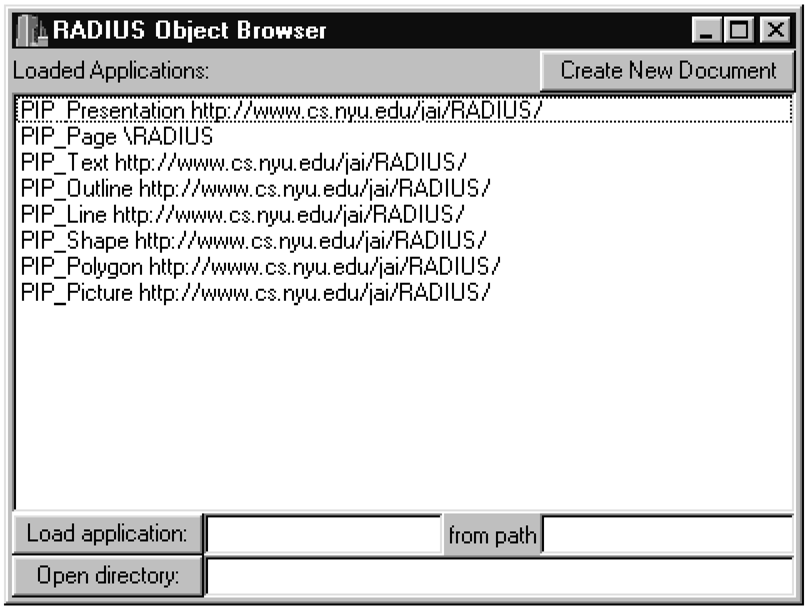
3.4. The Object Browser

The Object Browser is a skeletal user interface that can be used to start applications. We designed the user interface to be identical for the C++ and Java versions; therefore, we do not need to introduce them separately.



```
PIP.Component.Picture \Java
PIP.Component.Text \Java
```

Figure 21: RADIUS Object Browser in Java

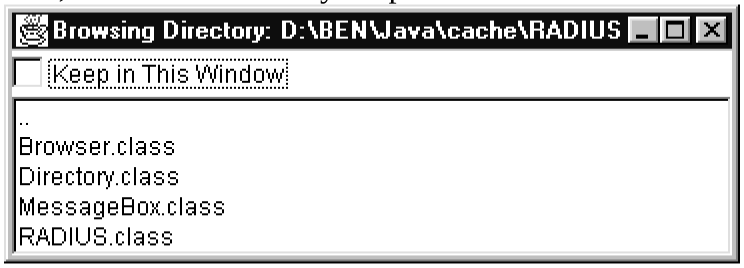


```
PIP_Presentation http://www.cs.nyu.edu/jai/RADIUS/
PIP_Page \RADIUS
PIP_Text http://www.cs.nyu.edu/jai/RADIUS/
PIP_Outline http://www.cs.nyu.edu/jai/RADIUS/
PIP_Line http://www.cs.nyu.edu/jai/RADIUS/
PIP_Shape http://www.cs.nyu.edu/jai/RADIUS/
PIP_Polygon http://www.cs.nyu.edu/jai/RADIUS/
PIP_Picture http://www.cs.nyu.edu/jai/RADIUS/
```

Figure 22: RADIUS Object Browser in C++

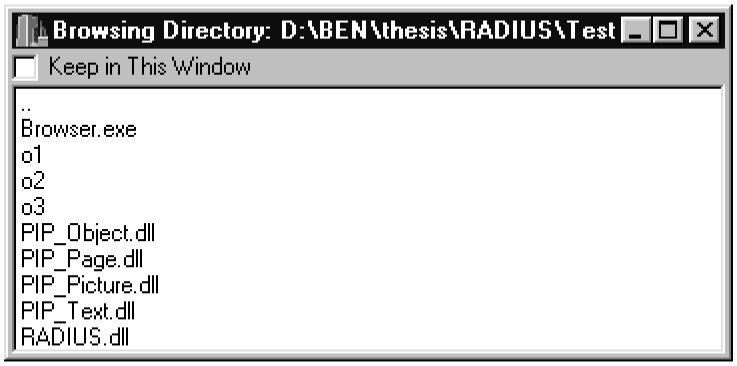
The list box in the middle lists one string for each of the applications that has been loaded into memory. The first part of the string is the Class Identifier and the second part is the Application Locator. When an item is double-clicked (or selected and then the "Create New Document" button is pressed), a new, blank document of the application is created.

An application is loaded into memory when a document of the application is opened through the Object Browser. Applications can also be explicitly loaded by typing the Class Identifier and a search path in the two edit boxes below the list box and then pressing the "Load Application" button. Finally, the "Open Directory" button opens a directory (which may reside over the network) specified in the edit box next to it. If the edit box is left blank, the current directory is opened.



```
..
Browser.class
Directory.class
MessageBox.class
RADIUS.class
```

Figure 23: A Directory Window in RADIUS – Java Version



```
..
Browser.exe
o1
o2
o3
PIP_Object.dll
PIP_Page.dll
PIP_Picture.dll
PIP_Text.dll
RADIUS.dll
```

Figure 24: A Directory Window in RADIUS – C++ Version

When the directory window is double-clicked, one of the following actions takes place:

1. If the user double-clicks on a directory, including the parent directory "..", the directory is opened. If the "Keep in This Window" checkbox is checked, the new directory is displayed in the current window, otherwise a new directory window is created.
2. If the user double-clicks on a file that is a RADIUS application, it is loaded into memory.
3. If the user double-clicks on a file that is a RADIUS document, it is loaded into memory and its `view` method is invoked, as illustrated in Figure 4.
4. Otherwise, an error message is displayed.

Case 1 provides means of navigating up and down the directory tree. Case 2 gives the user a way to load an application without a document (in addition to using the "Load Application" button). Case 3 is the standard method of opening RADIUS documents. It actually takes only one line of code (in addition to exception handlers) to display a document file once the name of the file is known.

```
Factory::createFromFile(filename)->view(); // C++
Browser.createFromFile(filename).view(); // Java
// in both languages: catch and handle exceptions
```

Figure 25: Code to Display a Document

The following figures show some sample applications using RADIUS. These samples are taken from a presentation authoring and programming system we built. The details of the system are presented in Chapter 4.

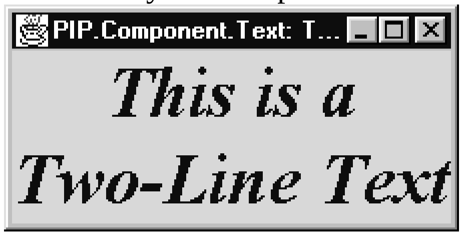


Figure 26: Text Application in Java

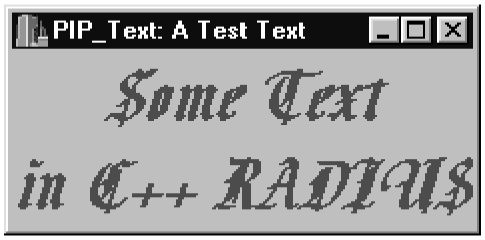


Figure 27: Text Application in C++

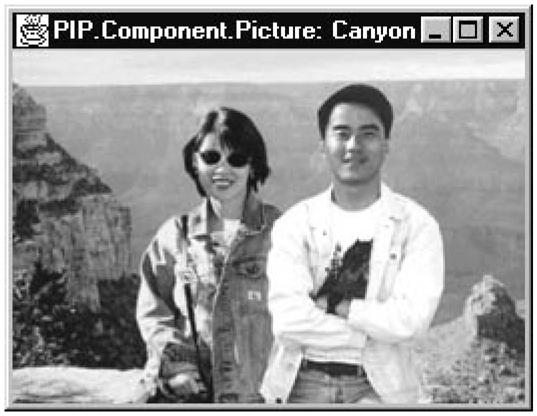


Figure 28: Picture Application in Java

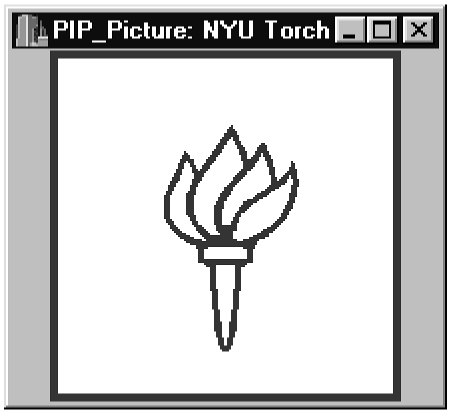


Figure 29: Picture Application in C++

The applications run in the same address space as the Object Browser. However, the applications do not need the Object Browser in order for them to be started. The C++ version of RADIUS has the Object Factory and Application Loader compiled into one single DLL file `RADIUS.DLL`. The Object Browser is an executable file that uses `RADIUS.DLL`. Applications can be designed to utilize `RADIUS.DLL` independently. Similarly, the Java version of RADIUS has the Object Factory coded in the Application Loader class `RADIUS.Loader`, which loads the class specified as the first command line argument upon startup. The Object Browser is compiled into `RADIUS.Browser` class and is started by the command line:

```
{path}java {-CLASSPATH} RADIUS.Loader RADIUS.Browser
```

Just replace `RADIUS.Browser` by any application class and the application will run without an Object Browser.

Chapter 4: Application – Programming in Presentation

Based on the RADIUS infrastructure, we built a programming environment and presentation-authoring system PIP (Programming in Presentation) which is a component-style extensible tool designed to simplify the construction of presentation-style programs. It features an easy-to-use user interface for building visual displays, yet with scripting capability, each element of a presentation can be turned into a full-fledged program. The developers can easily add new types of presentation element classes to the system through its RADIUS-based structure and programs built in PIP are automatically presentable. As in the case of RADIUS, PIP is implemented in both C++ on 32-bit Windows and in Java. Due to the lack of standard multimedia capabilities in Java, the C++ version is more powerful for building presentations and is therefore our current choice for demonstrations.

4.1. Motivation

Computers have changed a lot in physical dimensions during the past decade, enough for them to be widely used as tools for presentations. On the other hand, the explosive growth and broad penetration of the World Wide Web (WWW) has demonstrated the tremendous demand for the exchange of information. We have already seen many programs built solely for the purpose of information delivery running on notebook computers or Internet-connected desktop systems. Nevertheless, these programs are either under-powered or very hard to build. Most of the tools available concentrate on only one aspect of the problem – presentation or programming. If programs and presentations can be built together, a lot of work can be saved. As a response to this demand, we have designed and implemented PIP for developing presentation style programs.

4.1.1. Programming Models

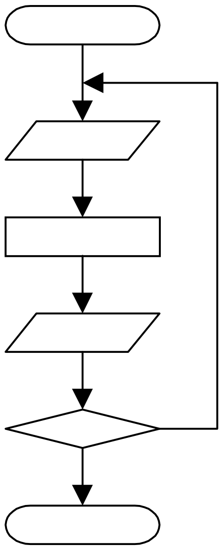


Figure 30: A Typical Sequential Program

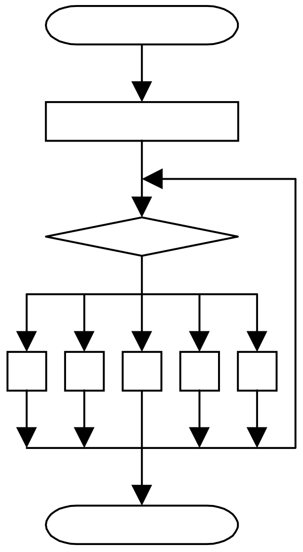


Figure 31: A Typical Event-Driven Program

Based on the structure of control flow, we can describe application programs as sequential or event-driven. Most of today's programming tools are designed to build programs exclusively in one of these two modes, even though the applications may use both in a mixed hierarchy.

In the early days of computers, programs worked in the sequential mode. A sequential program is a sequence of input and output steps [32, 36, 42]. Sequential programs are sometimes called "console mode" programs for this reason. The process may be repeated or constructed hierarchically, but the basic structure remains the same.

As the use of graphical user interface (GUI) systems became common, the event-driven style of programming matured into a new standard model. An event-driven program can be described as a collection of states whose transitions are triggered by events. Parts of an event-driven program get executed in an order not preset in the program, but following the occurrences of events. After necessary initialization, a GUI program usually enters an event loop from where the events are dispatched. Most modern GUI software development tools are designed to build programs in this form [5, 20, 34, 38, 40]. Many event-driven programs that have graphical user interfaces are designed to start with a main screen from where the user can go to other screens and back.

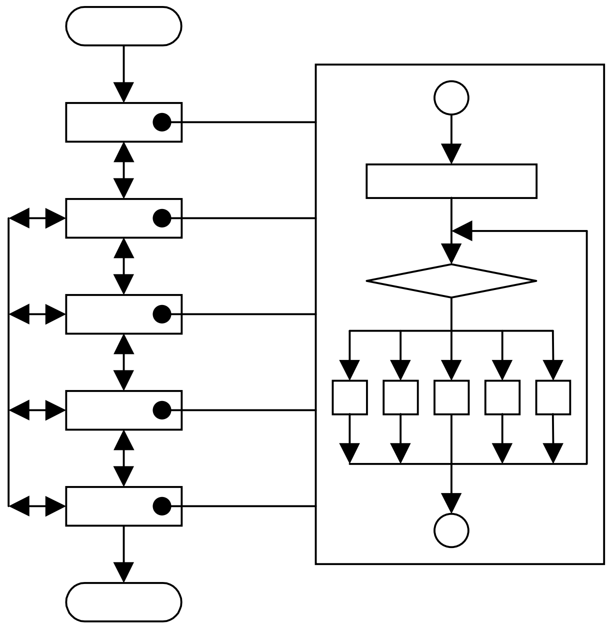


Figure 32: A Typical Presentation Style Program

Presentations need to have a control flow that is a mixture of sequential control and event-driven control. The steps of a presentation are arranged primarily in a linear fashion where as each step may contain event-driven activities. The steps in a presentation are also very often treated as states that can be transitioned to.

Building powerful presentation programs is often very awkward because existing tools typically support either sequential processing or a main-screen/sub-screen model of transition, but not both at the same time.

4.1.2. Related Work

The conventional way of presenting information is to build slides. A number of slide-based presentation systems (Microsoft PowerPoint™, Lotus Freelance™, Corel

Presentations™) have been on the market for some time. They let the users create complicated graphical displays with dazzling effects. However, they concentrate on the presentation chores and programming is not their strong suit. They provide some level of macro language support, but these constructs are insufficient and unsuitable for serious programming tasks. Multimedia software development tools (MacroMedia Director™, ToolBook™) are also effective in building lively presentations. While they concentrate on the media handling functionality and performance tuning, the scripting control languages suffer from the same shortcomings as in slide-based presentation tools.

Visual development tools for general purpose programming languages (Visual Basic, Visual C++, Visual J++, Delphi, C++ Builder, J Builder, Visual Café) provide the full power of the programming languages, but their aims are on the event-driven programming model. Substantial work has to be done to tailor their standard code templates for a presentation style control flow.

The World Wide Web was designed to be a medium for information exchange. It has been successful in delivering static and, to some extent, dynamic content. Indeed, many presentations have been constructed in HyperText Markup Language (HTML) format. However, the look and feel of HTML-based presentations is limited by the formatting capability of HTML, and the computing power is often restrained by security concerns of Web browsers.

4.2. The Basic Component

In addition to having persistent representations as all RADIUS objects do, PIP components also have visual representations and scriptable visual interactions. All PIP components derive from the same base class `PIP_Object`, which provides the basic implementation of PIP methods. We explain the design and implementation in this section.

4.2.1. The Data Structure

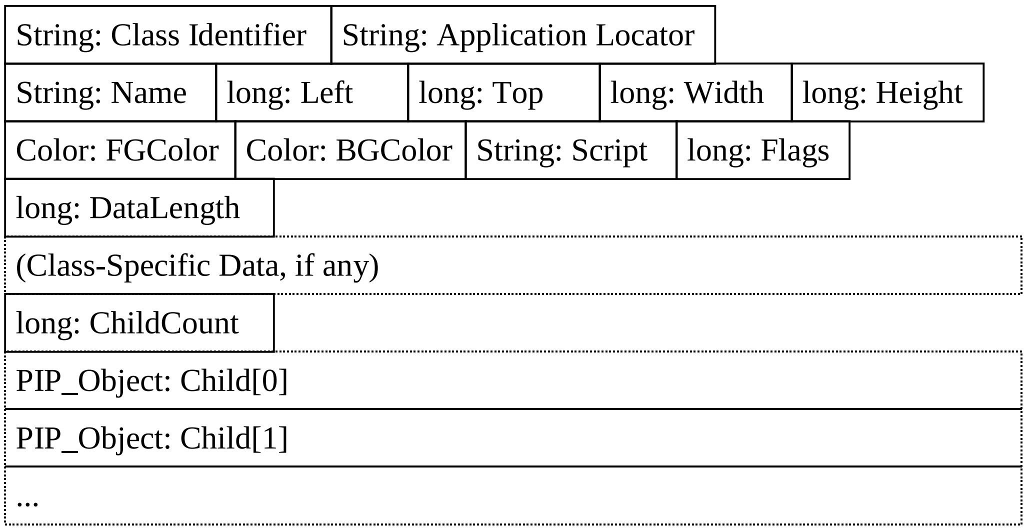


Figure 33: PIP File Structure

The above figure illustrates the persistent data structure of a `PIP_Object`. The Class Identifier and the Application Locator have been explained earlier. The remaining fields form the data portion of a standard RADIUS document. All PIP components stream to the same file structure and the only varying part is the class-specific data. By using a Data Length field to keep track of the size of the class-specific data, `PIP_Object` can provide a fallback implementation and maintain the data integrity when the application for the component cannot be found.

The `Name` field is used to identify the object to scripts. It can be an arbitrary string if the native script engine is used; otherwise, it has to adhere to the naming rules of the scripting language being used. The `Script` field stores the text form of the script program. The `Left`, `Top`, `Width`, `Height`, `FGColor` and `BGColor` fields are used in the visual representation to indicate the coordinates, dimensions, foreground and background colors. The `Flags` field stores various miscellaneous information about the object. In the base class `PIP_Object`, only one bit is used to indicate the visibility of the object.

When a `PIP_Object` is loaded into memory, the persistent data is read into corresponding fields. There are also some transient data fields, such as boolean flags to indicate whether the object is painted with a frame or is selected for editing, that are used only when the object is in memory. Accessor methods (in the form of `getFoo` and `setFoo`) are provided for all data fields, including the child objects.

4.2.2. The Data Methods

The following figure shows the methods in `PIP_Object` concerning the data of the objects, in addition to the RADIUS methods. (We designed all methods to be virtual, so the `virtual` modifiers are omitted here.)

```
PIP_Object *clone(void);
void cloneFrom(PIP_Object *obj);
void parseData(void);
void assembleData(void);
String getData(void);
bool setData(String data);

void clearChildren(void);
void addChild(PIP_Object *child);
PIP_Object *removeChild(long index);
void deleteChild(long index);
void moveChild(long f, long t);
PIP_Object *getChild(String name);
long getChild(long X, long Y);
long getIndex(void);
```

Figure 34: Data Methods in `PIP_Object`

The `clone` and `cloneFrom` methods provide copying operations. They are implemented with the streaming methods so they shouldn't need to be overridden even if a new component has special data. The `parseData` and `assembleData` methods are invoked from within the `readFromStream` and `writeToStream` methods, respectively, to process class-specific data. The `getData` and `setData` methods are used to access the "representative" data of an object in text form. The actual data field(s) accessed depend on the design of the class.

The `clearChildren` method removes all child objects. The `addChild` method adds a child object to the object. The `removeChild` and `deleteChild` methods remove a child object with or without returning it. The `moveChild` method changes the order of child objects. The getChild methods retrieve a child object by its name or retrieve the index of the child object whose display area contains the coordinates of a point. The getIndex method retrieves the index of an object within its parent object.

The parseData and assembleData methods should be overridden in new components in order to handle class-specific data.

4.2.3. The Visual Interface

```
void moveTo(long X, long Y);
void move(long dX, long dY);
void setDimension(long W, long H, long dir);
void shrinkWrap(void);

void setupDisplay(void);
void show(void);
void unShow(void);
void createPeer(void);
void destroyPeer(void);

void paint(TCanvas *c, RECT &pr, double zoom, double X, double Y);
void paintContent(TCanvas *canvas, RECT &maprect, double zoom);
void paintFrame(TCanvas *canvas, RECT &maprect);
bool render(double zoom);
long contains(long X, long Y);
void print(TCanvas *c, RECT &pr, double zoom, double X, double Y);
void print(TPrinter *p, long printmode);
void printContent(TCanvas *canvas, RECT &maprect, double zoom);

void markChanging(PIP_Object *child);
void unmarkChanging(void);
void markChanged(PIP_Object *child);
```

Figure 35: Visual Methods in `PIP_Object`

The visual representation of a PIP_Object can be manipulated by using the first set of the methods listed above (moveTo through shrinkWrap), in addition to using the accessor methods on the data fields. The move and moveTo methods are self-explanatory. The setDimension method uses an extra parameter (dir) to determine which corner or side to pin down when the dimension of the object is changed. The shrinkWrap method is used to adjust the dimension of an object according to the values of other data fields. For example, it is overridden in the PIP_Picture class to provide the functionality of resetting a picture to its natural size.

The second group of methods (setupDisplay through destroyPeer) is used to handle the internal parameters of the visual representation of the object. These methods are invoked automatically when needed. The setupDisplay method is called whenever a data field is changed in a way that affects the visual representation. It sets up display parameters, such as the bounding rectangle, of the object. The show and unShow methods are called when the object is physically added to or removed from the display. PIP objects are designed to be lightweight [35] in the sense that they perform their own visual rendering. However, sometimes it is reasonable to use components provided by third party or the operating system to save development effort. These reused components are called "peer" objects. The createPeer and destroyPeer methods are called from within the `show` and `unShow` methods, respectively, to create and destroy any peer objects used by the object. They should be overridden in component classes that use peer objects and the component developer is responsible to implement proper methods to maintain the coherence between the component object and the peer object.

The third set of methods (`paint` through `contains`) controls the actual rendering of the graphical image of the object. The `paint` method is called when the graphical image needs to be displayed. It first calls the `render` method to generate an in-memory copy of the image. If the `render` method returns true, then the in-memory copy of the image is transferred to the display, otherwise the `paintContent` method is invoked to render the image directly onto the display. The `render` method caches the in-memory copy of the image so that if it is called again before the object is changed, the actual rendering does not need to be performed again. The `paintFrame` method is used to draw a frame (not necessarily rectangular) around the object when the object is being edited. The frames are also used to indicate the selection status of the objects in the editors. The `contains` method is used to decide if a point falls in the object's display area. The return value is 0 if the point falls outside the object, 1 if inside the object, and 2 through 9 if the point falls on one of the eight "resizing handles" of the object when the object is being edited and is selected. The `printContent` method and the first `print` method are like the `paintContent` and `paint` methods, except that they render onto the printer instead of a display surface. They should be overridden when the object (e.g. a silhouette picture) requires a different rendering mechanism when sent to printers. The second `print` method sends the object to the printer as a complete document.

The last group of methods are used to inform the display which region of the object needs to be repainted. If a child object is given, only the area of the child object is refreshed. The `markChanging` method marks the area to refresh, but does not perform the refresh until the `markChanged` method is called, at which time the two refresh areas are joined together to form the actual refresh area. It is used when the modification of the object intertwines with the screen refresh process. For example, the `edit` method calls the `markChanging` method at the beginning and calls the `markChanged` method at the end so that both the old and new display areas are refreshed. The `unmarkChanging` method cancels the previous `markChanging`.

All components have two visual representations: one for design-time use, one for run-time use. Usually the only difference is that the design-time visual representation has a frame drawn around the component to indicate selection status and to provide a resizing border. This is achieved by the `paintFrame` method and the `Framed` and `Selected` internal flags (not listed). For non-visual components, the run-time visual representation is simply empty and the entire design-time visual representation is rendered by the `paintFrame` method. The `setFramed` method controls the internal `Framed` flag, which is checked by the `paint` method to decide if the `paintFrame` method should be invoked. The frame is displayed differently depending on whether the component is selected, which is indicated by the `Selected` internal flag controlled by the `setSelected` method.

Component developers should override the following methods when developing new components:

- Override `setDimension` if the dimensions of the component are subject to special constraints. For example, the component may have a minimum or maximum size, or has to maintain a certain aspect ratio.

- Override `shrinkWrap` to reflect the "natural" size of the component. For example, it sets the size of a picture component to the original size of the picture.
- Override `setupDisplay` to set up internal data structure for display, if the component requires more than just the bounding rectangle in order to define its display geometry.
- Override `createPeer` and `destroyPeer` if the component uses peer objects for display.
- Override `paintContent` to draw the image of the component, or override render to render an in-memory copy of the image.
- Override `paintFrame` and contains when the shape of the component (and therefore the frame) is not rectangular. For example, text at an angle.
- Override `printContent` if the component is rendered using a different mechanism when sent to a printer. The default implementation of `printContent` just passes all arguments to the `paintContent` method.

4.2.4. The Event Interface

```
bool mouseMove(long X, long Y, long shift);
bool mouseDown(long X, long Y, long button_shift);
bool mouseUp(long X, long Y, long button_shift);
bool click(long button_shift);
bool doubleClick(void);
bool mouseEnter(void);
bool mouseExit(void);
bool gotFocus(void);
bool lostFocus(void);
bool keyDown(long key, long shift);
bool keyPress(long key);
bool keyUp(long key, long shift);
String command(String cmd);
bool passEvent(bool propagate, EVENTTYPE event,
  long p1=0, long p2=0, long p3=0);
```

Figure 36: Event-related Methods in `PIP_Object`

The methods listed above are used to control the object's reaction to events. Instead of letting the underlying operating system pass events directly to the script engine, we defined the PIP event methods so that component developers can pre-process and post-process the events. For example, a button component uses the `mouseDown`, `mouseMove` and `mouseUp` methods to create the "pressed-down" look of the button and issue its own version of `CLICK` event; a list box component uses `mouseDown` and `doubleClick` methods to issue an "Item Action" event; a hot-spot component highlights the display region when the mouse cursor travels on top of the sensitive region.

The standard implementation of event methods just passes all events through the `passEvent` method, which in turn passes the events to the current script engine by invoking the `execute` method of the script engine (see next subsection). The boolean return value indicates whether the event has been processed. If an object does not process an event and the object has a parent object and the `propagate` flag was set to `true`, the event will be passed up to the parent object.

The `command` method responds to a special kind of event – user commands in the script. The return value indicates the error status. An empty string indicates that the command succeeded, otherwise the error message is contained in the returned string. This method provides a gateway for dynamically performing a function of the object without type information.

The `command` method should be overridden in new components if the component developers want to give users dynamic control of the component. The other methods should be overridden if the component needs to respond to events before passing them to the script engine.

Event Name	Description
`SHOW`	The visual representation of the object is about to be displayed.
`SHOWN`	The visual representation of the object has been displayed.
`UNSHOW`	The visual representation of the object is about to be destroyed.
`UNSHOWN`	The visual representation of the object has been destroyed.
`CHANGING`	The data of the object are about to change.
`CHANGED`	The data of the object have changed.
`PRINTING`	The object is about to be printed.
`PRINTED`	The object has been printed.
`EXCEPTION`	An exception has been thrown while running the object's script.

Figure 37: Other PIP Event Types

In addition to the events generated by the event methods listed in Figure 36, the events listed in the table above are also generated by the objects. The first eight events are designed to let the users customize the visual representations of the object at the instance level in addition to the class level. For example, the `PRINTED` event can be used to print a hidden note page attached to a presentation page; the `PRINTING` event can be used to print a banner page before the output; the `SHOW` and `UNSHOWN` events can be used to start and stop play the tape in a VCR; the `CHANGED` event can be used to trigger the recalculation of data in another object. The component developers can add new events to their new components. For example, a media player component may want to generate an event to indicate that the end of the media has been reached. As long as the `passEvent` method is invoked, the script engine will get a chance to handle the events.

Note that the viewing of the object is also an event, so even totally sequential programs that have no GUI interactions can be written in PIP by putting the script in the `SHOW` event handler of an object.

4.2.5. The Scripting Interface

Each object in PIP contains a piece of script that is stored in text form. PIP has a built-in native script engine, but the user can choose to use other script engines as well. The script engine needs to implement the interface listed in Figure 38. We plan to provide wrapper classes for the most popular script engines [7] such as VBScript, JavaScript and Tcl.

When an object is loaded into memory, its script gets compiled into an internal binary form to enhance the performance. This is achieved by the `compileScript` method. The `disposeCompiledScript` method disposes the internal binary form of the script. The script engine may choose to always interpret the script in text form and therefore not implement these two methods.

The `executeEvent` method is where the script gets executed. The first argument indicates the owner of the script and the second argument indicates the source of the event triggering the execution. These two objects may not be the same since the event may be passed from a child object to its parent object. The next four arguments are passed directly from the `passEvent` method of the source object.

The `editScript` method invokes an editor provided by the script engine on the script, which is passed in as the only argument. The other methods are self-explanatory.

```
class PIP_Script_Engine
{
public:
  virtual bool initialize(void)=0;
  virtual bool editScript(String &script)=0;
  virtual void compileScript(PIP_Object *scriptobj)=0;
  virtual void disposeCompiledScript(void *cs)=0;
  virtual bool executeEvent(PIP_Object *scriptobj, PIP_Object
    *eventobj, EVENTTYPE event, long p1, long p2, long p3)=0;
  virtual void enable(void)=0;
  virtual void disable(void)=0;
};
```

Figure 38: The `PIP_Script_Engine` interface

The relation between PIP components and the script engines can be illustrated in the following figure. The dotted arrow in the middle is where different script engines can plug in. The user can choose to use a different script engine for programming in a more familiar environment, or to use no script engines at all for designing plain presentations without programming.

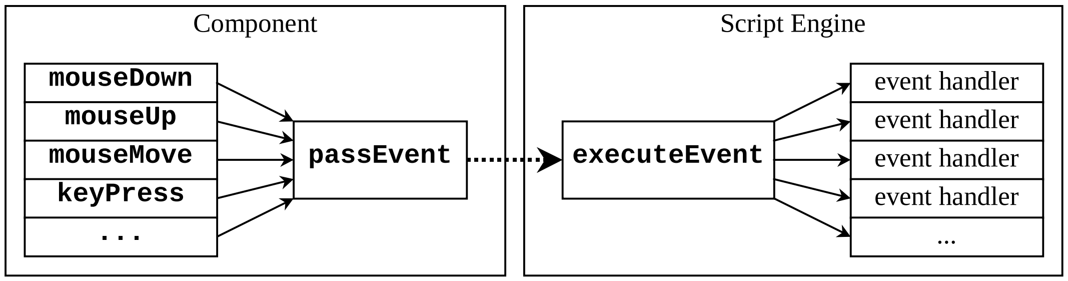


Figure 39: PIP Event Model

Instances of script engines are associated with viewing windows, not objects. The object being viewed in a viewer window and all the descendent objects share one instance of the script engine and thus can pass information among themselves.

4.2.6. Viewing and Editing PIP Objects

When a PIP document is opened in the RADIUS Object Browser, the object is loaded into memory and the `view` method is invoked. The standard `view` method performs the following steps:

- The `setupDisplay` method of the object has been invoked in the `readFromStream` method to prepare the object's transient properties for visual display.
- A standard object viewer window (of class `PIP_Object_Viewer`) is created with a display component (of class `PIP_Display`) occupying the display area. The viewer window creates an instance of the native script engine and associates the engine with

the display component.

- The object and the display component are associated with each other.
- The show method of the object is called to create the visual representation. It generates a SHOW event, calls the createPeer method to create peer objects, if any, calls the show method of all child components, and generates a SHOWN event.
- The paint method of the object is called when the visual representation of the object is actually painted on screen. It calls the paintContent method or the render method to display the body of the component.

If a component should not be viewed as a stand-alone application, the view method can be overridden to do nothing. The view method can also be overridden to perform customized viewing behavior. For example, the view method of the PIP_Presentation class starts the presentation in a designer instead of a viewer.

The standard viewer window accepts user commands such as changing the display zoom ratio and edit or print the object. When the viewer window is closed, the following sequence takes place:

- If the object has been modified, a very common "save" query sequence takes place to confirm if the user wants to save the object to the document.
- The unShow method of the object is called to destroy the visual representation. It generates an UNSHOW event, calls the unShow method of the child objects, calls the destroyPeer method to destroy any peer objects that have been created, and generates an UNSHOWN event.
- An UNLOAD event is generated, the child objects are deleted and the object itself is deleted. This step itself is recursively applied to all child objects when they are deleted.
- The script engine, the display component and the viewer window are deleted.

```
bool view(void);
PIP_Object_Editor *editor(void);
bool edit(void);
PIP_Object_InPlace_Editor *inPlaceEditor(void);
bool inPlaceEdit(void);
```

Figure 40: Viewing and Editing Methods in PIP_Object

If the object can be viewed stand-alone, the view method should start viewing the object and then return true. The view method returns false if the object cannot be viewed stand-alone or there was an error viewing the object.

When an edit command is issued in the standard viewer window of an object, the edit method is invoked. The standard implementation uses an editor window (of class PIP_Object_Editor) returned by the editor method to let the user modify the attributes of the object. The edit method returns true if the object has been changed in the editor. It is also invoked when an object is being edited as a child component in a presentation page.

Sometimes the editing of an object has better visual look-and-feel to the users if the editing takes place on the display surface instead of in a separate editing window (so that the change is immediately visible to the users). This is usually called "in-place editing." In such cases, the inPlaceEdit method will be used. It overlays an in-place editor object (returned by the inPlaceEditor method) on top of the object being displayed and lets the user manipulate the object directly. This feature is not being used in our current implementation.

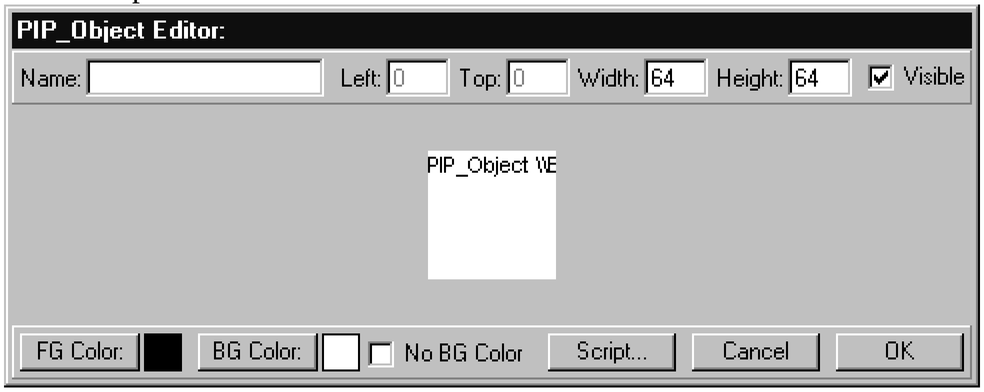


Figure 41: Standard Editing Window for `PIP_Object`

Each component class creates its own customized editor by adding control objects to the standard editor instead of deriving new editor classes from `PIP_Object_Editor`. The `editor` method returns an editor object with the extra control objects already added. The choice of using delegation here instead of inheritance was made just for the coding convenience in our development environment. The Java version of PIP use inheritance for editor classes.

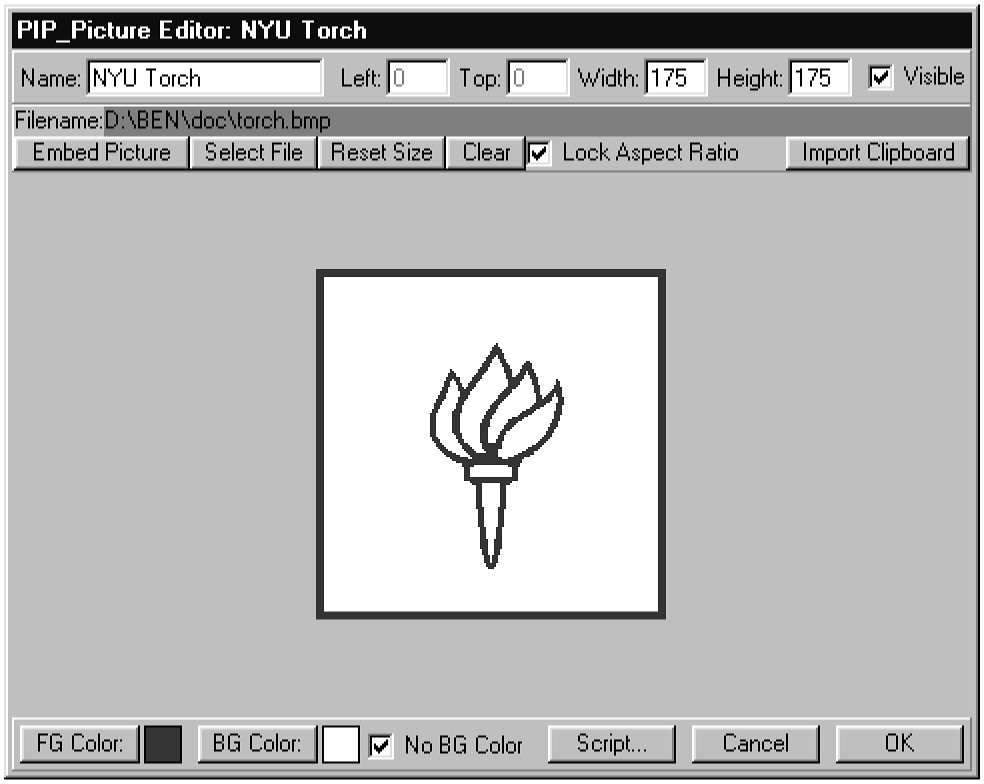


Figure 42: Customized Standard Editing Window for `PIP_Picture`

4.3. Standard PIP Components

In this section, we show how PIP components can be built from standard components.   Each of the following subsections shows and demonstrates the customization of the base class `PIP_Object` in different aspects.

Unless specifically noted, all component classes add some properties and override the `parseData`, `assembleData`, `paintContent` and `editor` methods. They provide accessor methods to the new properties and initialize the properties in the constructor. If a component class allocates some system resource for internal use, the resource will be freed in a virtual destructor. We will omit the explanation of these common customizations.

4.3.1. The PIP_Text Class

The `PIP_Text` class represents a block of text. The text may be drawn transparently if the background color is set to none. New properties include `Font`, `Size`, `BIUS` (Bold, Italic, Underline and Strikeout style flags), `Align` and `Text`. The `shrinkWrap` method is overridden to set the dimension of the text block to the minimal rectangle containing the text.

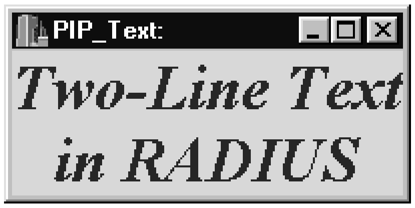
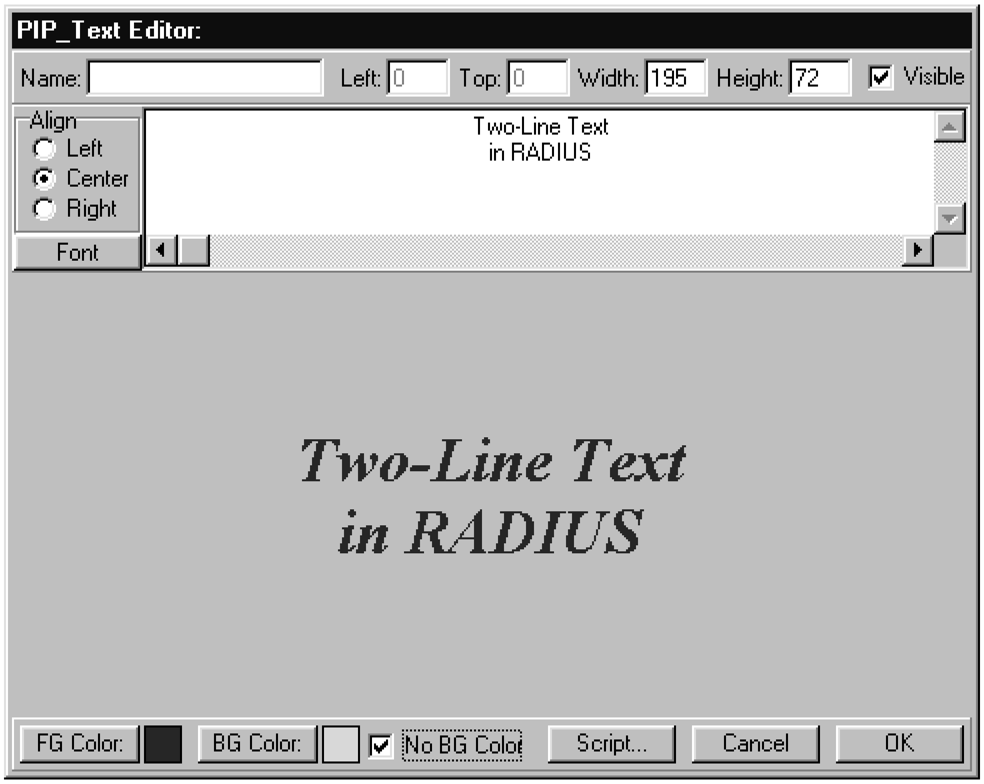


Figure 43: A `PIP_Text` Object



Name:
Left: 0 Top: 0 Width: 195 Height: 72 [✓ Visible]

Align: Left [✓ Center] Right

Two-Line Text
in RADIUS

[✓ No BG Color]

Figure 44: Editing a `PIP_Text` Object

4.3.2. The PIP_Picture Class

The `PIP_Picture` class represents a picture in Windows Bitmap, Windows Metafile or GIF format. The actual picture data may be contained in a separate file ("linked") or stored in the data portion of the object ("embedded"). The picture can be set to maintain a constant aspect ratio and/or have a transparent color. New properties include `Format` (BMP, WMF or GIF), `Linked`, `LockAspect` and `PictureFileName`.

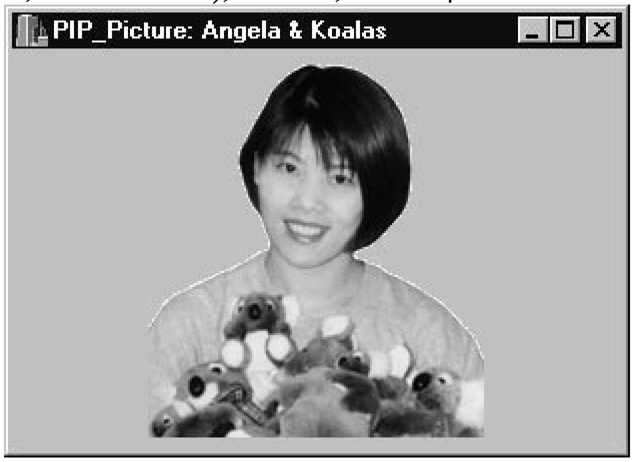
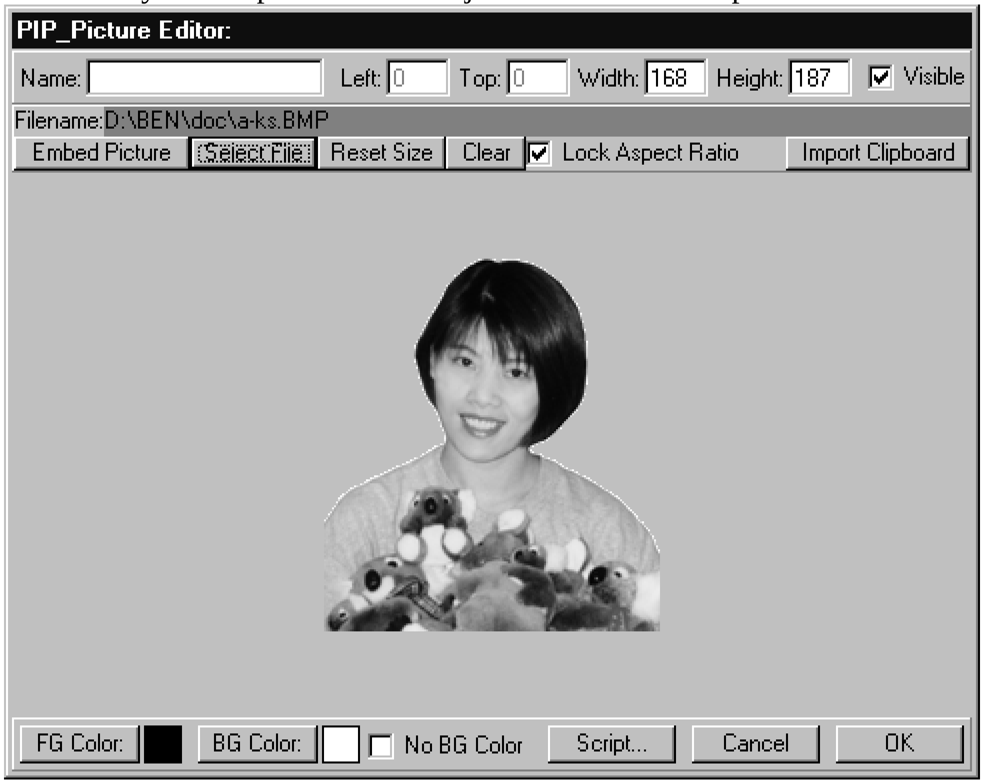


Figure 45: A `PIP_Picture` Object

The `shrinkWrap` method is overridden to set the dimension of the picture block to the original picture size. The `setupDisplay` and `setDimension` methods are overridden to observe the locked aspect ratio. The `render` method is overridden to generate a mask for the transparent color. The `printContent` method is overridden to render the picture to printers. Special processing is necessary to render a color image onto a black-and-white printing device. A new method `importClipboard` is added to transfer the picture contained in the system clipboard to the object as an embedded picture.



Name: Left: 0 Top: 0 Width: 168 Height: 187 [✓ Visible]

Filename: D:\BEN\doc\a-ks.BMP

[✓ Lock Aspect Ratio]

No BG Color

Figure 46: Editing a `PIP_Picture` Object

The property `BGColor` has different meanings according to the format of the picture. For Windows Bitmap pictures, it indicates which color is not to be displayed; for Windows Metafile pictures, it is the color used to fill the background before rendering the Metafile; for GIF pictures, it has no effect. Transparent colors for GIF pictures are specified in the data. If the picture is empty, the object is shown as a solid rectangle filled with `BGColor`. Furthermore, for monochrome Windows Bitmap pictures, the `FGColor` property specifies which color is used to replace black. Therefore, the same black-and-white picture can be used in different places in different colors.

4.3.3. The PIP_Page Class

The `PIP_Page` class is a component that utilizes the child component features of `PIP_Object` to compose compound objects. It is derived from the `PIP_Picture` class. The picture serves as the background of the page on top of which the child objects are displayed. When a page is contained in a presentation and there is no picture nor background color for the page, the presentation's background will be used.

For editing purposes, the `view` method is overridden to start the page document in a designer window instead of a viewer window. While in the designer, the child objects are displayed with frames to indicate their boundaries and selection status. Menu commands and direct-manipulation in the display area provide a user interface to handle child objects, including adding, deleting, moving, resizing and aligning them. A command is provided in the designer to allow the page to be shown in a regular viewer window.

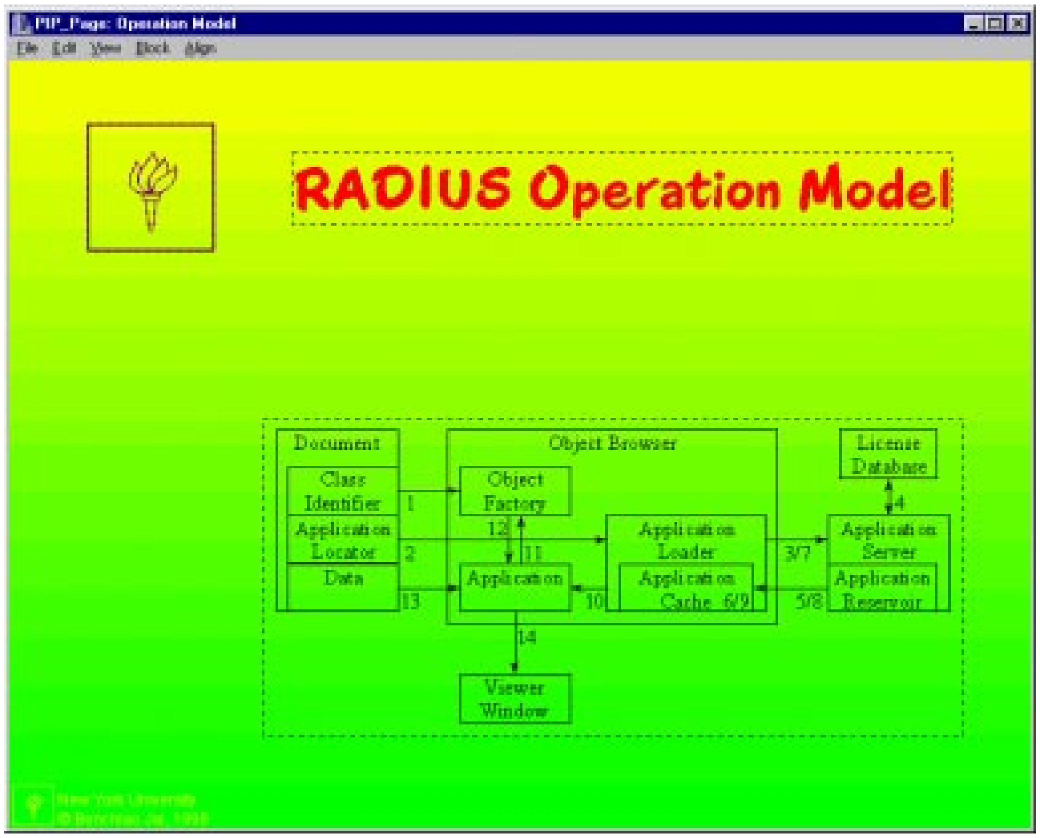


Figure 47: PIP Page Designer

The `PIP_Page` class also serves as a "grouping" container for combining multiple objects into one. Grouping and ungrouping commands are provided in the designer for this feature.

4.3.4. The PIP_Presentation Class

The `PIP_Presentation` class represents presentations composed of pages. It is derived from the `PIP_Page` class. The `PIP_Page` portion of a `PIP_Presentation` object serves as the "master slide" of the presentation. The child objects are the actual pages displayed. The pages in a presentation are not necessarily shown in a linear fashion. Most likely, they will be hyper-linked like Web pages.

The `view` method is overridden to start the presentation document in a designer window, just as in the case of `PIP_Page`. `PIP_Page` and `PIP_Presentation` actually use the same designer program, which dynamically responds to different types of objects being designed. Additional menu commands are provided to handle pages. A "play" command starts showing the presentation.

4.3.5. The PIP_Line, PIP_Shape and PIP_Polygon Classes

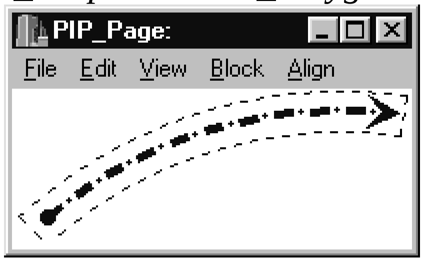
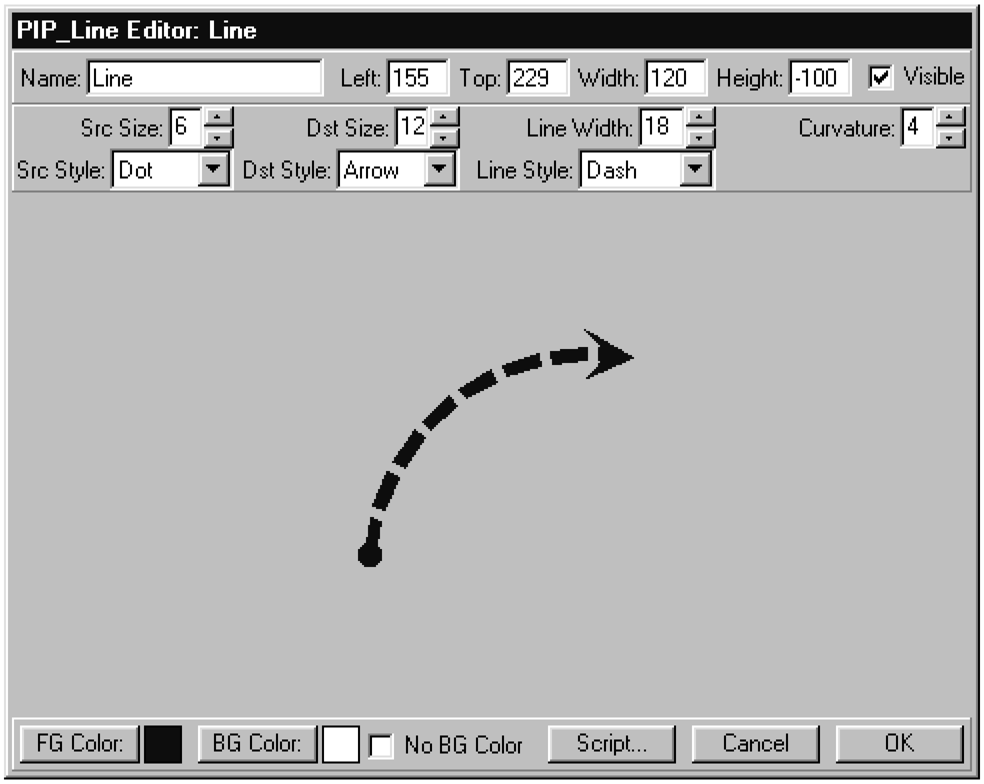


Figure 48: A Curved Frame for a `PIP_Line` Object



PIP_Line Editor: Line

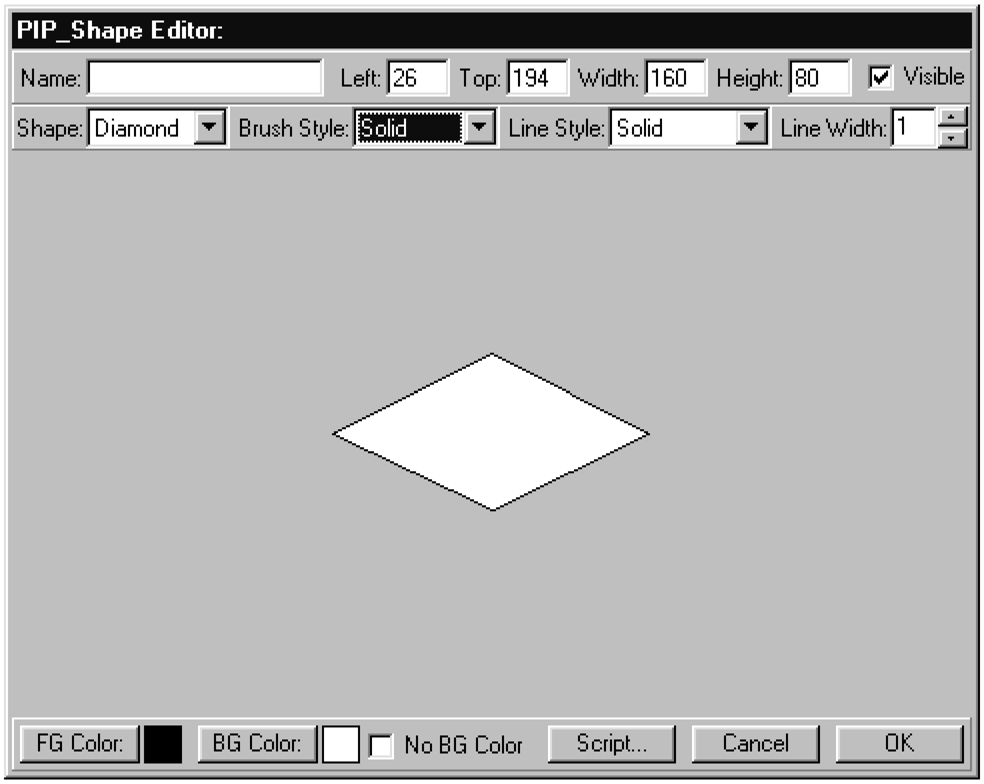
Name: Line Left: 155 Top: 229 Width: 120 Height: -100 [✓ Visible]

Src Size: 6 Dst Size: 12 Line Width: 18 Curvature: 4

Src Style: Dot Dst Style: Arrow Line Style: Dash

No BG Color

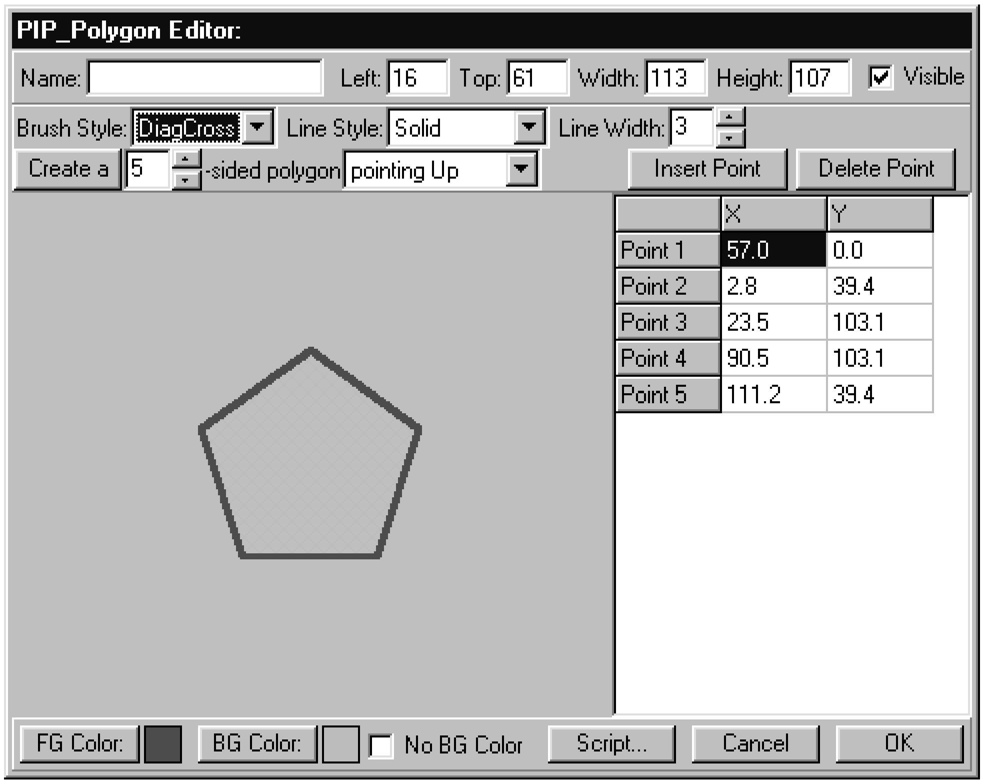
Figure 49: Editing a `PIP_Line` Object


PIP_Shape Editor:

Name: Left: 26 Top: 194 Width: 160 Height: 80 ☑ Visible

Shape: Diamond Brush Style: Solid Line Style: Solid Line Width: 1

Figure 50: Editing a `PIP_Shape` Object


PIP_Polygon Editor:

Name: Left: 16 Top: 61 Width: 113 Height: 107 ☑ Visible

Brush Style: DiagCross Line Style: Solid Line Width: 3

Create a 5 -sided polygon pointing Up

	X	Y
Point 1	57.0	0.0
Point 2	2.8	39.4
Point 3	23.5	103.1
Point 4	90.5	103.1
Point 5	111.2	39.4

Figure 51: Editing a `PIP_Polygon` Object

The `PIP_Line` class demonstrates how PIP object can have non-rectangular display regions. The line can have a dashed, dotted, or other styles and the endpoints can be circles, arrowheads, squares or triangles. The `setupDisplay` method is overridden to set up the internal data structure of the frame shape and end point polygons. The `paintFrame` method is overridden to draw the non-rectangular frame. The `contains` method is overridden to test whether a point falls in certain regions of the line object. It works differently at design-time and run-time. At design-time the entire region within the frame is counted as inside the object, at run-time only the line itself counts.

The `PIP_Shape` and `PIP_Polygon` classes are simple objects representing geometric shapes. They are also useful in specifying "clickable maps" when they are laid on top of a picture with their colors set to transparent.

For the time being, a `PIP_Shape` object can be an ellipse, a rectangle, a rectangle with round corners, a diamond shape or a flowchart terminal symbol. In the future, more shapes may be added and the `PIP_Shape` class is likely to be upgraded often to include the new shapes.

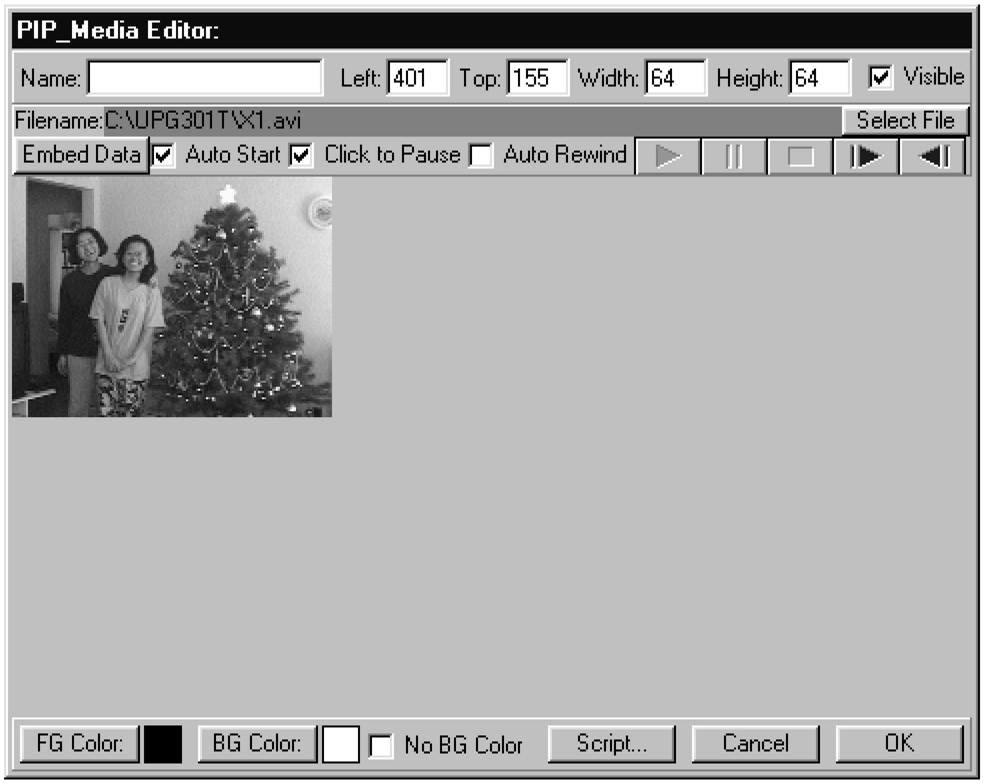
4.3.6. The PIP_Timer Class

The `PIP_Timer` class is a non-visual component. New properties include `Interval` and `Enabled`. It introduces a new type of event – `TIMEOUT` – which is issued at the user-specified interval if the timer is enabled. The `paintContent` method is overridden to not paint anything. The `paintFrame` method is overridden to paint the design-time visual image. The `contains` and `setDimension` methods are overridden to work at design-time only and to prevent resizing of the visual image. The `createPeer` and `destroyPeer` methods are overridden to handle the system timer peer object, which is non-visual.

4.3.7. The PIP_Media Class

The `PIP_Media` class is a visual component with an invisible visual peer class. The peer class encapsulates a Windows Media Player, whose visual representation contains the control buttons we set to invisible. The visible form of the buttons can be seen near the top-right corner of the following figure. If the media clip contains video frames, the video is shown in the display area of the `PIP_Media` object. Acceptable media formats include Windows Video (.avi), Audio (.wav) and MIDI Sequencer (.mid, .rmi) files. The `paintContent` method is overridden to just set up the video playing rectangle without painting anything. The `paintFrame` method is overridden to paint the design-time visual image. The `createPeer` and `destroyPeer` methods are overridden to handle the system media player peer object. The `command` method is overridden to accept commands "`Play`", "`Stop`", "`Pause`", "`Resume`" and "`Rewind`." This class also issues the `TIMEOUT` event when the media clip finishes playing.

Three very often-used features are designed as properties, although they can be achieved by scripts. They are `AutoStart` (starts playing as soon as the object is shown), `ClickToPause` (mouse click on top of the object will pause/resume playing) and `AutoRewind` (automatically rewinds after playing is finished). The "Embed Data" button can be used to package the media clip data with the object, as in the case of `PIP_Picture`.


PIP_Media Editor:

Name: Left: 401 Top: 155 Width: 64 Height: 64 [✓ Visible]

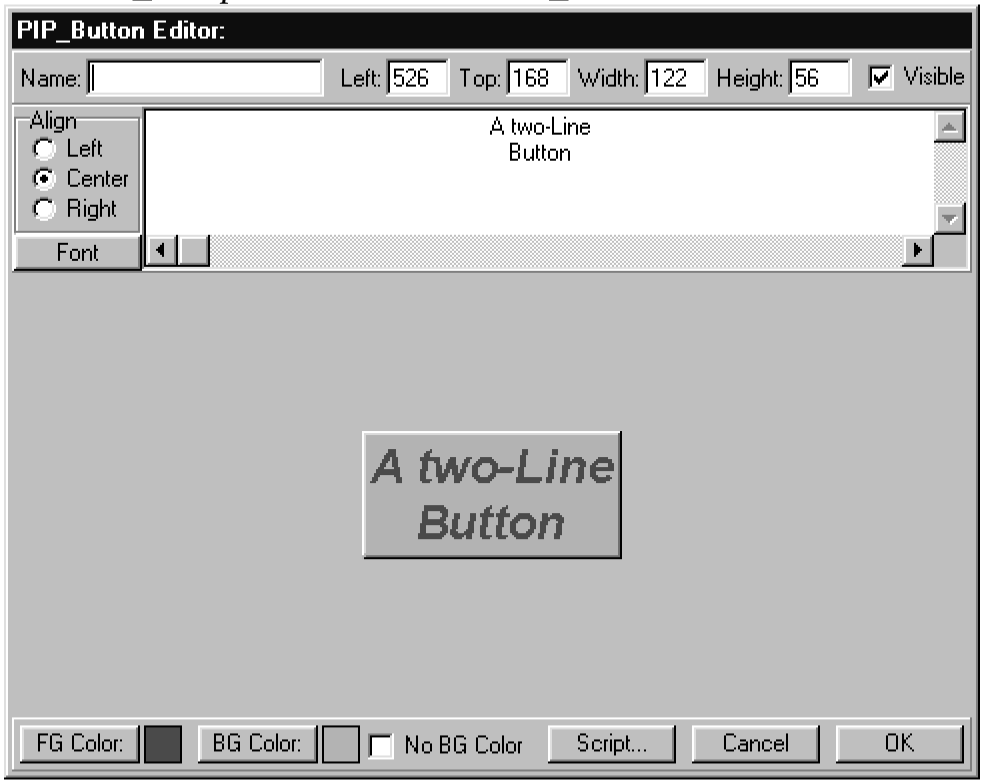
Filename: C:\UPG301T\X1.avi

[✓ Auto Start] [✓ Click to Pause] Auto Rewind

FG Color: BG Color: No BG Color

Figure 52: Editing a `PIP_Media` Object

4.3.8. The PIP_Hotspot Class and the PIP_Button Class


PIP_Button Editor:

Name: Left: 526 Top: 168 Width: 122 Height: 56 [✓ Visible]

Align: Left [✓ Center] Right

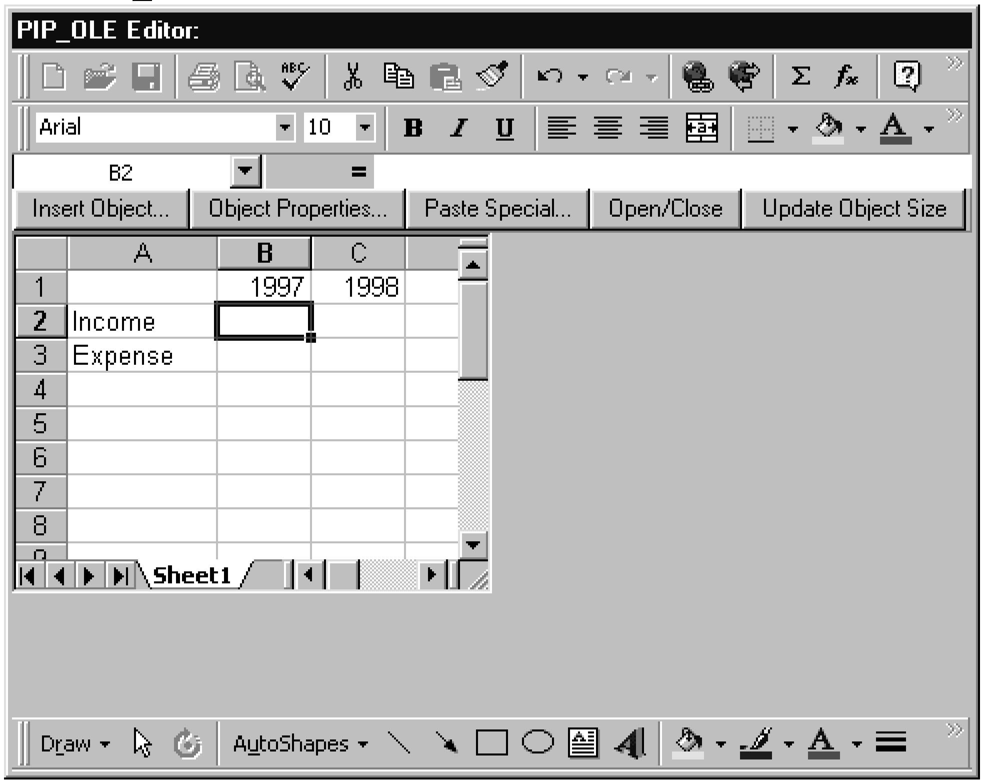
A two-Line
Button

FG Color: BG Color: No BG Color

Figure 53: Editing a `PIP_Button` Object

These two classes demonstrate components that perform pre-processing of system events. The `PIP_Hotspot` class overrides the `mouseEnter` and `mouseExit` methods to change the visual image when the mouse cursor moves over the display area. The `PIP_Button` class overrides the `mouseMove`, `mouseDown` and `mouseUp` methods to generate the "pressed-down" look. Other than these behavioral differences, they are just subclasses of `PIP_Picture` and `PIP_Text` with different visual representations.

4.3.9. The PIP_OLE Class



	A	B	C
1		1997	1998
2	Income		
3	Expense		
4			
5			
6			
7			
8			

Figure 54: Editing a `PIP_OLE` Object Encapsulating an Excel Spreadsheet

The `PIP_OLE` class enables the users to incorporate OLE (Microsoft's Object Linking and Embedding) objects into their presentations. Essentially any OLE application on the users' computers can be reused this way. However, since OLE applications may not be self-installing, the presentation may not be able to display the OLE object when it is moved to a different computer.

The `PIP_OLE` class uses a visible visual peer class, which is an OLE container that works between the display surface and the OLE server applications. The `createPeer` and `destroyPeer` methods are overridden to handle the peer object. The `paintContent` and `printContent` methods are overridden to pass the rendering tasks to the server applications. The `command` method is overridden to pass commands to the server application.

4.3.10. The PIP_Engine Class

The `PIP_Engine` class implements the native script engine of PIP. It compiles the scripts of PIP objects into a binary form that runs on a very simple virtual machine. The intrinsic instructions of the virtual machine contain only code execution, value cloning, address calculation and storage management. All other functions are achieved by plug-in verb modules. A more detailed description of the internals of the script engines can be found in the next section.

The `PIP_Engine` class is a RADIUS class. The DLL file for the class is installed automatically by the object viewer and the presentation player. The `PIP_Engine` class, in turn, automatically installs a standard verb module. If the `PIP_Engine` class or the standard verb module are somehow not installed, the presentations can still be viewed in the traditional slide show fashion. The scripts just don't get executed in that case.

4.3.11. The PIP_Table and PIP_Chart Classes

Tables and charts are common ways of presenting data. The `PIP_Table` class represents matrices of data. The entries can be manually filled-in or generated by programs. The `PIP_Chart` class represents graphical display of data stored in table format. The number of types of charts may increase as new ones are developed. This is another example of an application that may be upgraded often.

4.3.12. The PIP_Database Class

In the business world, database (or the current buzzword "data warehouse") is not just a convenience, it is a necessity. Every large corporation has its own huge collection of data from which they want their people to build presentations. We do not intend to write a big powerful database management system to cater to every possible database. Instead, we took a standardized approach to provide access to databases through ODBC and SQL. The `PIP_Database` class allows users to specify database parameters and then issue queries. The results are returned in a table property that can be linked directly to charts or processed by scripts.

4.4. PIP Script Programs

PIP components are programmed at two levels. The behavior pertinent to a component class is programmed as class methods by the component developers. The behavior customized for an instance of a component class is programmed as scripts by the component users, who in turn are the presentation developers. The final presentations are used by presentation presenters. Here the term "presentation" is used loosely to indicate any PIP objects being viewed, not just those of class `PIP_Presentation`.

PIP scripts are executed in script engines. PIP has a built-in native script engine which is light weight and fast, but the developers may choose to use other script engines through the standard programming interface provided by PIP. In this section, we limit our discussion to the native script language PIPScript.

PIPScript has a very simple syntax. We define the script text attached to a PIP object as a "program." A program consists of a set of variable declarations, a set of event handlers and a set of user-defined functions. Variables can be local to an event handler/user-defined function, local to the program or global to the script engine. All variables have the same type – object – although some internal optimization is implemented for numerical and string values. An event handler consists of a name and a command. A user-defined function consists of a name, an optional list of input arguments, an optional list of output arguments and a command. User-defined functions are used exactly the same way as verbs. A command consists of a verb and a set of nouns, or a variable declaration, or a sequence of commands enclosed in a pair of curly braces. A noun can be a variable, a literal value or a command that returns values. A literal can be a number or a string. All numbers are specified in the textual format of double precision floating numbers and are internally stored as such, although during computation they may be converted to integral or boolean types. String literals follow the standard C string specification. Comments start with double slash and runs to the end of the line. They can appear anywhere in the program.

Program ::= Unit*
Unit ::= Global_Variable_Declaration | Event_Handler | User_Defined_Function
Global_Variable_Declaration ::= [`global` | `var`] Name* `CR`
Event_Handler ::= `on` Name Command
User_Defined_Function ::= `function` Name Name* [=> Name*] `CR` Command
Command ::= Verb Noun* | Variable_Declaration | `{` Command* `}`
Variable_Declaration ::= [`global` | `var` | `local`] Name* `CR`
Noun ::= Variable | Literal | Command
Literal ::= Number | String

Figure 55: PIPScript Grammar Rules

The syntax of PIP commands is in a way like LISP without parenthesis. Nevertheless, one feature not commonly found in other languages is that a function or command can return more than one value. As long as the enclosing command accepts enough arguments, all return values of a command will be used. Unused return values are just discarded.

All words in PIPScript are separated by white spaces. Line breaks are insignificant except in comments, variable declarations (to mark the end of the variable list) and function declarations (to mark the end of the argument list). There are only three reserved words: `{` and `}` for grouping commands and `//` for comments. Six other keywords have special meanings only if they appear at the right places: `global`, `var`, `local` (variable declaration), `on` (starts an event handler), `function` (starts a user-defined function) and => (separate the input and output parameter lists in the header of a user-defined function); otherwise they can be used as regular identifiers. At compile time, a word not in the set of reserved words will be first looked up as a user-defined function, then looked up as a verb, then tested for literal, then looked up in the variable tables (local, program and global). An identifier (for both the nouns and the verbs) is a string that does not form a literal. For example: `7-Eleven`, `A&P`, `2*Pi` are all legitimate identifiers.

PIPScript is designed to have full access to all information in PIP objects. It also has the facilities to start an executable program, load a library module and invoke external methods to handle tasks that cannot be implemented in PIPScript.

Programmers can write their own verbs in C++ and compile them into DLL modules, which are loaded at run-time. The verb modules have control over both the compile-time and run-time behavior of the verbs. PIPScript comes with a set of standard verbs, including object property manipulation, mathematical and logical functions, program flow control, list operations, string operations, and utility functions. These standard verbs are packaged into one verb module.

```
// script program in a PIP_Page object
var deltaX deltaY ball screenW screenH Module
on SHOW
{
  set deltaX 5
  set deltaY 8
  set2 screenW screenH
    getDimension this // getDimension returns two numbers
  set ball getChild this "ball"
  set Module loadLib "DBViewer" // external library
  call Module "open" "stock.DB" // external method
}
// move the top-left corner of the ball to the point clicked at
on MOUSEDOWN setLocation ball param1 param2
// bounce the ball within page
on TIMER
{
  move ball deltaX deltaY
  if > getBoundsRight  ball screenW set deltaX –5
  if < getBoundsLeft   ball 0       set deltaX 5
  if > getBoundsBottom ball screenH set deltaY –8
  if < getBoundsTop    ball 0       set deltaY 8
}
on DOUBLECLICK call Module "view" 0 // external method
on UNSHOWN freeLib Module

// script program in a PIP_Shape object
// An animator module adds the new event type ANIMATIONEND and
// the new verbs animate, animating and stopAnimate.
// A user defined function.  The object is moved to a new random
// location with a new random size through 10 animation steps.
function jump
{
  local w h
  set w + 10 * random 200
  set h + 10 * random 200
  animate this // the object to animate
    * random - getWidth getParent this w
    * random - getHeight getParent this h // new location
    w h // new dimension
    10 // number of steps
}
on CLICK if animating this stopAnimate this else jump
on ANIMATIONEND jump // go again
```

Figure 56: Sample PIPScript Program

4.4.1. Security for PIP Objects

Since PIP objects have script programs attached and the script engines are built in the RADIUS framework, each PIP object can be viewed as a piece of mobile code. When the receiver of a PIP document opens the document, the RADIUS facility will install all relevant applications and starts the execution of the programs contained in the document. This is a very powerful and potentially dangerous mechanism that can easily be exploited by malicious programmers. Although the applications can be authenticated at the binary level, the script programs can still be used to do harm to the client computers. This same phenomenon has been observed in Microsoft Word™, whose scripting language VBA is powerful enough to be used in developing viruses.

Fortunately, we can learn from the development experience of Microsoft Word and the Java applet security mechanism. Facilities similar to theirs can be employed in the script engine to alert the users if a change is to be made to the computer, or simply lock out local resources from the script programs. Furthermore, code authentication can also be provided at the script level. These will be part of our future work.

4.5. Summary

Software development is a creative process. Programming models and patterns are usually identified long after they have been used in practice. In this chapter, we described Presentation Style Programming as a programming model that is becoming popular. Programs need to be presented, and presentations need to be smarter. We can foresee that more programs will be written in presentation style in order to deliver information in better ways.

We also presented a software development environment designed to aid the creation and maintenance of presentation style software. Our system is flexible and easy to use. On one extreme, lay users can easily build slides for pure presentation; on the other extreme, advanced programmers can easily develop heavy-duty programs with the full power of C++; in the middle, a convenient scripting facility is provided for an easy way to build programmed presentations, presentable programs, and customized presentation systems for use by lay users.

Chapter 5: Conclusion

In this dissertation we developed the RADIUS application framework to simplify the tasks of building software that is self-installing and self-upgrading, by extending the object-oriented programming paradigm from the programming language (data/code) level to the operating system (document/application) level. This philosophy can be easily understood by imagining the Internet as a huge address space where pointers are symbolic.

Another important idea is revealed in the unified view of content delivery and software deployment. By treating application code as another form of content to be delivered, software deployment is achieved by mechanisms similar to those for information delivery.

While attaining the intended design goal, software built in RADIUS requires minimal work on the developers' part and absolutely no work on the users' part. One important attribute of RADIUS is that it achieves its objectives without requiring any new technologies to extend programming languages and operating systems. The very efficient design and low coding overhead are collateral benefits of our high-level abstractions. We believe that this characteristic should enhance the acceptance and potential usage of this system.

We have also demonstrated how application components can be easily implemented and assembled in the RADIUS environment. There are many ways for RADIUS to be used, and there is some practical work that should be done to enhance the usefulness of this system. We now address these issues in detail.

5.1. Application Areas of RADIUS

RADIUS is intended to be used wherever software deployment overhead is high. The amount of such work is directly proportional to the number of users and the frequency of revisions, and of course, the amount of work involved in each installation. The last issue does not really restrict which situations are suitable for RADIUS. We believe that the design principles of RADIUS can be applied to all software to reduce the amount of deployment work needed in each installation.

According to the first two factors mentioned above, we can classify applications into four categories as shown in the following table:

Case	Number of users	Number of revisions	Applicable category
1	Large	Large	Information subscription, content delivery, shareware
2	Large	Small	COTS (Commercial Off-The-Shelf) software
3	Small	Large	Intranet applications
4	Small	Small	Custom software for lay users

Figure 57: RADIUS Application Areas

Although software licensing, security and code authentication mechanisms will need to be added before RADIUS can be used in an open environment, the current system is already suitable for use in Intranet, shareware and custom applications where software licensing and code authentication either are non-issues or can be handled by other facilities. Security is a more platform- and application-dependent issue that is orthogonal to the philosophy and design of RADIUS. We plan to leave it for the individual programming systems to handle.

5.2. Future Directions

With RADIUS as a basic framework for application software, we can incorporate other technologies to extend the interoperability among applications and reduce the learning curve for both the users and the developers.

5.2.1. Generalizing RADIUS

Our two implementations of RADIUS are constructed in two programming languages, yet they can be viewed as constructed in two operating systems, considering the unique characteristics of Java. The following figure shows the possible combinations between programming languages and operating systems. It is fairly easy to generalize RADIUS to other platforms by using the same techniques in our current implementations.

Programming Languages / Operating Systems	Java	C++	Ada	SmallTalk	Eiffel	Others...
Windows	✓	✓	★	★	★	★
Unix	✓	★	◆	◆	◆	◆
Linux	✓	★	◆	◆	◆	◆
MacOS	✓	★	◆	◆	◆	◆
OS/2	✓	★	◆	◆	◆	◆

✓: RADIUS has been implemented.
★: RADIUS can be implemented as a direct translation of an existing RADIUS.
◆: Needs further investigation.

Figure 58: Current Status of RADIUS Implementations

The task of generalizing RADIUS can be approached from two directions: programming languages or operating systems. For programming languages that are highly independent of the underlying operating systems, it will be easy to build one implementation of RADIUS and use it on all operating systems. However, it is likely to be less work and more general to build one RADIUS core for each operating system and provide RADIUS functionality to the programmers as a standard extension to the operating system. The latter approach also implies the possibility of binary data compatibility among applications built with different programming languages, which is an important characteristics of popular component systems (e.g. COM, CORBA).

5.2.2. Integrating the Object Browser

Instead of running as a stand-alone application, the Object Browser can be integrated into a Web Browser so that the Internet can become an extended file system. By viewing the Application Loader as a linker/loader from the operating systems aspect, we see that RADIUS can even be considered as part of an operating system. The Object Browser can be integrated into the command shells of existing operating systems to enhance their function. As the line between Web Browsers and command shells become thin (as in the case of Internet Explorer in Windows 98), these two scenarios may actually merge into one in the near future.

Furthermore, the documents can have the Object Browser bundled into them so that they become documents that install their own applications. Currently the users still need to download/copy the Object Browser and the RADIUS runtime library to start using RADIUS. No matter how small these files are, it is still one step that the users have to perform. We can easily package all the files necessary for RADIUS into one utility program that, when executed, reads in a document and attaches itself to the document to produce a RADIUS-installing document. This philosophy has been employed in utility programs like self-extracting archive builders.

5.2.3. The Storage and Exchange Format

Applications can be developed in different programming languages, on different operating systems or even on different hardware architectures. Rather than letting each application control its own data layout in a private manner, a data description language such as XML [52] can be used to standardize the external data format. After all, the programming languages, operating systems and even hardware architectures get aged and phased out, but the data lives on. For (a somewhat extreme) example, today's computers are still processing data collected by the Viking probes, and when the Voyager probes encounters the next intelligent life form, none of today's computers and software will still be in use.

If a binary standard format is to be used for storage space or communication bandwidth considerations, External Data Representation Standard (XDR) [44] (also used by HTTP-ng Binary Wire Protocol [19]) would seem to be a good candidate, although it is less decipherable from a human perspective.

The W3C (World-Wide-Web Council) has a standard Document Object Model (DOM) [53] describing an object-oriented programming interface for HTML (DHTML, XML, SGML) documents. DOM can also be used to map RADIUS objects to and from documents in a standard way.

If the storage format of RADIUS documents is unified across programming languages and operating systems, we can even consider using a unified Object Browser that automatically chooses among different versions of the same application implemented on different platforms to process the document. For example, a numerical document prefers a C++ version of the application for better efficiency, but if one were not available for the current operating system, a Java version might be acceptable.

5.2.4. COM/CORBA Interfaces

Applications developed in RADIUS are naturally component-based. It is quite easy to add the basic COM or CORBA interfaces so that they can interact with non-RADIUS component systems. Furthermore, the basic RADIUS programming interfaces can be extended to include standard services of COM or CORBA so that developers can easily build COM/CORBA objects with RADIUS functionality.

5.2.5. Software Licensing

Software licensing is an important issue that needs to be addressed for COTS (Commercial Off-The-Shelf) software. The requests sent by the Object Browser provide a basic mechanism for license control. The application server can challenge the object browser for digital licenses, or simply look up a database for registered users. We have not yet implemented a licensing database mechanism; however, it is not an extensive endeavor. The database can be a simple relational database consisting of user/machine identifiers (e.g. user name, machine IP address or the latest CPU ID in the Pentium III™ chip) and software right descriptions (e.g. initial download and upgrades within one year, unlimited upgrades or three upgrades, etc.). The licensing mechanism will also need a software purchase process and maybe a license transfer process in case the users change machines.

Since the object browser also asks for upgrades, RADIUS even enables developers to catch software piracy. If the machine is unlicensed for the application, a warning can be issued and actions can be taken, as long as the actions are within the bounds of the law. Of course, the user can always disconnect the computer from the Internet to avoid being detected, but then the application may be useless anyway if it was designed to rely on the Internet (e.g. Mail, Chat, Internet Game, information delivery, etc.).

The more sophisticated pirates will be able to fake their identities and spoof a simple license check, but there are quite a number of techniques in cryptography that we can use to build a stronger verification protocol.

5.2.6. Security and Authentication

Our current design does not have any security or authentication measures, which makes it insufficient for use in developing commodity software. A few things need to be added in order for RADIUS software to be considered trustworthy. This can be addressed in two aspects: security for being free from benign errors, and authentication for being free from malignant errors.

Security requires more platform-dependent solutions since it relies heavily on the facilities provided by the operating systems[*], and sometimes even hardware. Therefore it is harder to find a uniform solution.

Code authentication is easier to achieve in a platform/language independent way. The application modules can be digitally signed by the developers and a verification protocol can be enacted between the Application Loader and the Application Server. If the network being used is in a secure communication environment, the authentication process only needs to be performed once each time there is a new application or upgrade. For example, in a secure LAN environment where all the machines load their applications from a proxy server, only the proxy server needs to perform authentication when there is an application loaded from outside the LAN.

5.2.7. RADIUS Applications as Mobile Code

Applications such as PIP can, in turn, be programming environments themselves. “Documents” for these applications are programs. When these documents are distributed, a form of distributed computing occurs. Since RADIUS applications are self-installing, this kind of distributed computing environment is very lightweight and does not require the programming/runtime tools to be pre-installed. With proper security and authentication mechanisms implemented and automated, we can envision RADIUS being used as an efficient and seamless rapid development tool for loosely-coupled distributed computing.

[*] Java can be considered part of an operating system for this issue.

BIBLIOGRAPHY

1. 20/20 Software. *Software Installation, Distribution and Security Solutions*. http://www.twenty.com/.
2. Apple Computer, Inc. *OpenDoc*. http://opendoc.apple.com.
3. Barnes, Donnie. *RPM HOWTO*. http://metalab.unc.edu/LDP/HOWTO/RPM-HOWTO.html.
4. Beveridge, Jim. *Self-Registering Objects in C++*. Dr. Dobb's Journal #288, August 1998, pp38-41.
5. Borland. *Borland JBuilder User's Guide*. Borland International, 1998.
6. Box, Don. *Essential COM*. Addison Wesley, Menlo Park, California, 1998.
7. Box, Don. *Say Goodbye to Macro Envy with Active Scripting*. Microsoft Interactive Developer, February 1998, Microsoft Press. Also available at http://www.microsoft.com/mind/0297/activescripting.htm.
8. Derfler, Frank, et al. *TKO Your TCO Today*. PC Magazine October 21, 1997, pp223-228.
9. Dykstra, Dave, and Katherine Lato. *NSBD and Software Distribution*. Dr. Dobb's Journal #289, September 1998, pp84-88.
10. Easter, Leslie. *Bulletproof Installs: A Developer's Guide to Install Programs for Windows*. Prentice Hall, 1998.
11. Flanagan, David. *Java in a Nutshell*. O'Reilly, Sebastopol, California, 1997.
12. Gamma, Erich, et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts, 1994.
13. Gilder, George, *The Coming Software Shift*. http://www.agribiz.com/fbFiles/readings/softshif.html.
14. Gosling, James, et al. *The Java Language Specification*. Addison-Wesley, Reading, Massachusetts, 1996.
15. Grimes, Richard, et al. *Beginning ATL COM Programming*. Wrox Press, Birmingham, UK, 1998.
16. Hall, Richard S. *Software Deployment Information Clearinghouse*. http://www.cs.colorado.edu/users/rickhall/deployment.
17. Jai, Benchiao. *A Lightweight Dynamic OOP Framework for Automatic Application Location, Installation and Upgrade*. Proceedings of OOPSLA '98. ACM, New York, 1998. Pre-Addendum pp34-35.
18. Jai, Benchiao. *RADIUS: A Rapid Application Delivery, Installation and Upgrade System*. Proceedings of TOOLS Pacific '98. IEEE, Los Alamitos, California, 1998. ISBN 0-7695-0053-6. pp180-186.
19. Janssen, Bill, *HTTP-NG Binary Wire Protocol*. W3C, http://www.w3.org/TR/WD-HTTP-NG-wire.
20. Johnson, E. F., Reichard, K. *Power Programming MOTIF, 2ed*. MIS Press, 1993.
21. Liang, Sheng, and Gilad Bracha. *Dynamic Class Loading in the Java™ Virtual Machine*. Proceedings of the OOPSLA '98. ACM, New York, 1998, pp36-44.
22. Lindholm, Tim, and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, Massachusetts, 1997. Also available at ftp://ftp.javasoft.com/docs/specs/vmspec.html.zip.
23. Lockhart, Harold W. *OSF DCE*. McGraw-Hill, New York, New York, 1994.

24. Marimba, Inc. *Marimba Castanet™*. http://www.marimba.com/three.
25. McAfee. *Oil Change™ Online Manual*. http://www.cybermedia.com/support/oilchange/manual/index.html.
26. Microsoft. *The Component Object Model Specification*. http://www.microsoft.com/COM/COM1598B.ZIP.
27. Netscape Communications Corporation. *Using JAR Installation Manager for SmartUpdate*. August 27, 1997. Mountain View, California.
28. Netscape Communications Corporation. *SmartUpdate for Content Developers*. http://developer.netscape.com/docs/manuals/communicator/jarforcd/index.htm.
29. OMG, *The Common Object Request Broker: Architecture and Specification, V2.2*. OMG, ftp://ftp.omg.org/pub/docs/formal/98-02-01.pdf, 1998.
30. OMG. *CORBAfacilities: Common Facilities Architecture*. OMG, ftp://ftp.omg.org/pub/docs/formal/corbafacility-97-06-15.pdf, 1997.
31. OMG, *CORBAservices: Common Object Services Specification*. OMG, ftp://ftp.omg.org/pub/docs/formal/corbaservice-97-12-02.pdf, 1998.
32. Orr, K.T. *Structured Systems Development*. Prentice Hall, 1977.
33. Pal, Partha Pratim. *A Flexible, Applet-like Software Distribution Mechanism for Java Applications*. Software Engineering Notes Vol. 23 No. 4, July 1998, pp56-60.
34. Parker, R.O. *Easy Object Programming for MacIntosh Using AppMaker and Think C*. Prentice Hall, 1992.
35. Perelman-Hall, David K. *Java and Lightweight Components*. Dr. Dobb's Journal #296, February 1999, pp22-28.
36. Pressman, R. S. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, 1982.
37. Price Waterhouse. *Technology Forecast: 1998*. Price Waterhouse Global Technology Centre. Menlo Park, California. 1998.
38. Prosise, J. *Programming Windows 95 with MFC: Create Programs for Windows Quickly with the Microsoft Foundation Class Library*. Microsoft Press, 1996.
39. Raymond, Eric S. *The Cathedral and the Bazaar*. http://www.tuxedo.org/~esr/writings/cathedral-bazaar/cathedral-bazaar.ps.
40. Reisdorph, K. *Teach Yourself Borland C++ Builder 3 in 14 Days*. Sams Publishing, 1998.
41. Sessions, Roger. *COM and DCOM: Microsoft's Vision for Distributed Objects*. John Wiley & Sons, New York, 1997.
42. Stay, J.F. *HIPO and Integrated Program Design*. IBM Systems Journal, 15, 2, 1976, pp143-154.
43. Stroustrup, Bjarne, *The C++ Programming Language, Third Edition*. Reading, Massachusetts: Addison-Wesley, 1997.
44. Srinivasan, Raj, *XDR: External Data Representation Standard*. Internet RFC 1832, http://info.internet.isi.edu:80/in-notes/rfc/files/rfc1832.txt.
45. Sun Microsystems. *JavaBeans™*. ftp://ftp.javasoft.com/docs/beans/beans.101.pdf.
46. Sun Microsystems. *Java™ Core Reflection. API and Specification*. ftp://ftp.javasoft.com/docs/jdk1.2/java-reflection.pdf, 1997.
47. Sun Microsystems. *Java™ Object Serialization Specification*. ftp://ftp.javasoft.com/docs/jdk1.2/serial-spec-JDK1.2.pdf, 1998.
48. Sun Microsystems. *Java™ Remote Method Invocation Specification*.

ftp://ftp.javasoft.com/docs/jdk1.1/rmi-spec.pdf.
49. Tallman, Owen H. *Project Gabriel: Automated Software Deployment in a Large Commercial Network*. Digital Technical Journal Vol. 7 No. 2, 1995, pp56-70.
50. Venners, Bill. *Inside the Java Virtual Machine*. McGraw-Hill, New York, New York, 1998.
51. Vlissides, M. and Mark A. Linton. *Unidraw: a framework for building domain-specific graphical editors*. ACM Transactions on Information Systems Vol. 8 No. 3, July 1990, pp237-268.
52. W3C. *Extensible Markup Language (XML) 1.0*. http://www.w3.org/TR/1998/REC-xml-19980210.html.
53. W3C. *Document Object Model (DOM) Level 1 Specification*. http://www.w3.org/TR/1998/PR-DOM-Level-1-19980818/DOM.ps.
54. Windows Magazine. *By the Numbers – Life Cycle Costs*. Windows Magazine May 1999, p28.