

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI[®]

THE UNIVERSITY OF ALBERTA

Design and Implementation of a Tool to

Support Web-based Dynamic Composition of Software Component

by

Raymond Wai-Man Wong



A THESIS

SUBMITTED TO

THE FACULTY OF GRADUATE STUDIES AND RESEARCH

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

EDMONTON, ALBERTA

SPRING, 2001



National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisitions et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-60515-9

Canada

THE UNIVERSITY OF ALBERTA

Library Release Form

Name of Author: Raymond Wai-Man Wong

Title of Thesis: Design and Implementation of a Tool to Support Web-based
Dynamic Composition of Software Components

Degree: Master of Science

Year this Degree Granted: 2001

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purpose only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as herein before provided, neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.



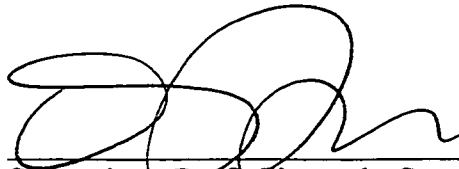
Raymond Wai-Man
Wong
942 Ormsby Wynd,
Edmonton, AB
T5T 6A9

April 17, 2001

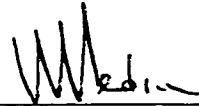
THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES

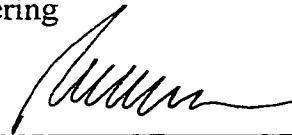
The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled "Design and Implementation of a Tool to Support Web-based Dynamic Composition of Software Components" submitted by Raymond Wai-Man Wong in partial fulfillment of the requirements for the degree of Master of Science.



Supervisor, Prof. Giancarlo Succi
Department of Electrical and Computer
Engineering



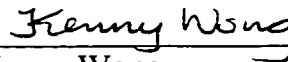
Prof. Witold Pedrycz
Department of Electrical and Computer
Engineering



Prof. Peter Musilek
Department of Electrical and Computer
Engineering



Prof. Marek Reformat
Department of Electrical and Computer
Engineering



Prof. Kenny Wong
Department of Computing Science

APRIL 17, 2001

Date

Abstract

Internet is a worldwide resource for component distribution. Software companies already sell their products to customers through direct downloading. This distribution model can be extended to software components instead of full-featured applications. Components are composed by customers to reform customizable applications. However, the current available technology does not support the distribution mechanism fully. This research work focuses on the study and development of a prototype for supporting the described market structure. In order to support the infrastructure, dynamic component composition in a Web-based environment must be supported. WebCODS (Web-based Component On Demand System) is a research prototype that supports dynamic component composition over the Web. This system uses the component composition model derived from UniCon and applies the composition framework in a Web-based environment. The framework also provides a classification model and a search engine to support component brokerage. The classification model is based on facet classification and accompanied with a search engine for querying the component broker. The security aspect in the environment is also addressed using encryption and digital signature. The thesis contains detailed discussion on the design of WebCODS and the selected implementation strategy. From the experience of developing WebCODS, the techniques for building such a system and directions for future research are identified.

Acknowledgements

This work has benefited from the support of many people. Thanks to my supervisor, Giancarlo Succi, for his timely advice, encouragement, and criticism throughout the development of this work. Thanks are also due to the people at the SERN lab at the University of Calgary and the QUASE Lab at the University of Alberta. These people have been an endless source of inspiration and good discussion throughout the work. They include Eric Liu, Skylar Lei, Mike Smith, Milorad Stefanovic, and Jason Yip.

I could not have completed this research without the support of University of Alberta, the University of Calgary, the Government of Alberta, the Alberta Software Engineering Research Consortium, and the Canadian Natural Sciences and Engineering Research Council.

This work is dedicated to my Parents, Kenneth Wong and Rita Lau.

Table of Contents

Chapter 1:	Introduction	1
1.1	Objectives	5
1.2	Thesis Structure	7
Chapter 2:	Component Composition.....	9
2.1	Scripting Languages	9
2.1.1	Module Interconnection Languages.....	10
2.1.2	Architectural Description Languages	11
2.2	Other Composition Approaches	16
2.3	Dynamic Component Composition Tools.....	18
2.4	Summary	21
Chapter 3:	Component Classification Background.....	24
3.1	Representation of Components	24
3.2	Software Libraries	28
3.3	Summary	32
Chapter 4:	Network Security Background	33
4.1	Keys and Certificates	34
4.2	Encryption Technique	36
4.3	Summary	38
Chapter 5:	JavaSpaces.....	41
5.1	Overview	41
5.2	Operations	43
5.3	Benefits.....	46
Chapter 6:	WebCODS.....	48
6.1	Client Environment	49
6.2	Provider Environment	50
6.3	Broker Environment.....	50
Chapter 7:	Support for Dynamic Composition	52
7.1	Framework Level	53

7.2	Composition Level	54
7.2.1	Ports and Connectors	56
7.2.1.1	Procedures Calls	57
7.2.1.2	Pipes (Streams).....	58
7.2.1.3	Events	59
7.2.1.4	Shared Variables	60
7.2.1.5	Files	60
7.2.1.6	Sockets	61
7.2.1.7	Supporting Connection Mechanisms	62
7.3	Instance Level	64
Chapter 8:	Component Brokerage.....	67
8.1	Component Classification	69
8.2	Operations of the Broker	72
8.3	Security.....	74
Chapter 9:	Implementation.....	76
9.1	Composition Tool.....	77
9.1.1	Interpreter.....	77
9.1.2	Transfer of Components	81
9.1.3	Composition Environment.....	83
9.2	Search Engine.....	87
9.3	Security Center.....	92
9.4	Fault Tolerance.....	96
Chapter 10:	Example Use of WebCODS.....	99
10.1	Providers.....	99
10.2	Broker.....	103
10.3	Clients.....	104
10.4	Integration of Existing Applications into WebCODS.....	106
Chapter 11:	Conclusions and Further Research.....	112
References	117
Appendix A:	WebCODS User Manual.....	124
A.1	Broker	125

A.1.1	Database File.....	126
A.1.2	INI File.....	127
A.1.3	Keystore File.....	127
A.1.4	Facets File	128
A.1.5	Thesaurus File.....	129
A.1.6	Execution of the Broker.....	131
A.1.7	Recovery Service	132
A.2	Client.....	133
A.2.1	Prerequisite to Execute the Application.....	133
A.2.2	Execution of the Application	135
A.2.3	Saving and Reloading of Components.....	138
A.3	Provider.....	141
A.3.1	Prerequisite to Execute the Application.....	141
A.3.2	Execution of the Application	144
A.3.3	Creation of Components	145
A.3.4	Classification of Components	148
A.3.5	Components in the Broker	149
Appendix B:	Description of the Implementation of WebCODS	150
B.1	Connector.....	151
B.2	Service.....	155
B.3	Parser.....	158
B.3.1	Creation of Primitive Components	158
B.3.2	Creation of Composite Components.....	159
B.3.2.1	Instance Section	160
B.3.2.2	Connectivity Section.....	161
B.4	Typing.....	163
B.4.1	Resolving of Primitive Components.....	167
B.4.2	Resolving of Composite Components	168
B.4.3	Textual Description.....	169
B.5	Broker	171
B.5.1	Data Access.....	172
B.5.2	Sessions in Server	174

B.5.3	Delivery Service.....	176
B.5.4	Security Center.....	177
B.5.5	Service Center	179
B.6	Search Engine	181
B.7	Client Application.....	184
B.7.1	Initialization	184
B.7.2	UI Creation.....	185
B.7.3	Compose Diagram	187
B.7.4	Searching and Downloading Components from the Broker	190
B.7.4.1	Searching.....	190
B.7.4.2	Downloading.....	191
B.8	Provider Application.....	192
B.8.1	Initialization	192
B.8.2	UI Creation.....	193
B.8.3	Creation Diagram.....	196
B.8.4	Compose Diagram	198
B.8.5	Submitting Components to the Broker	199

List of Tables

Table 1: Comparisons of supported features of component composition in ADLs.....	15
Table 2: Investigated software composition tools	21
Table 3: Summary of classification scheme	27
Table 4: Software libraries and their classification scheme.....	31
Table 5: Phases of software architecture analysis.....	55
Table 6: Summary for connections mechanism for components in Java.....	56
Table 7: Connectors in WebCODS.....	62
Table 8: Types of Ports supported in WebCODS.....	63
Table 9: Samples of Facets and Term Spaces used in WebCODS.....	71
Table 10: The result set after querying the graph with the term Parsing.....	91
Table 11: Options used in the <code>dx2j3d</code> component.....	107
Table 12: Icons for ports in WebCODS.....	146
Table 13: Package structure within the <code>connector</code> package	152
Table 14: Summary for the implementation of connectors	153
Table 15: Summary for steps required to create <code>TypedComponent</code> objects.	162
Table 16: Usage of supporting classes in the <code>typing</code> package	165
Table 17: Summary of the sessions created in the broker	174
Table 18: Supporting classes for drawing in the composition diagram.....	189
Table 19: Icons for ports in WebCODS.....	197

List of Figures

Figure 1: Sample Entry from the HelloWorld example	43
Figure 2: Sample code for writing and reading the HelloEntry from the space	44
Figure 3: Structure of WebCODS	48
Figure 4: Software Components and Connectors	53
Figure 5: Instantiation of Primitive Component	64
Figure 6: Instantiation of Composite Components	65
Figure 7: Component Brokerage Environment in WebCODS	68
Figure 8: Thesaurus for Action Facet	73
Figure 9: Security protocol used in WebCODS	74
Figure 10: Distribution of components in WebCODS	76
Figure 11: Connectivity description for the components shown in Figure 4	80
Figure 12: UML diagram shows the structure of components in WebCODS	81
Figure 13: Possible Java code for the connection of A and B in Figure 4	84
Figure 14: Template used by the Non Buffered Connector	85
Figure 15: The DTD for the XML description of WebCODS component	88
Figure 16: A sample XML description for WebCODS components	89
Figure 17: Example of relations in the Thesaurus	90
Figure 18: Transferring of keys between clients and the broker	94
Figure 19: Code for decryption and encryption in WebCODS	95
Figure 20: Protocols for transferring messages in WebCODS	96
Figure 21: Code for restoration of the broker	98
Figure 22: Creation of (a) primitive and (b) composite components in WebCODS	101
Figure 23: Generated Description for UniCon-based Description in WebCODS for (a) CParser, (b) Preprocessor and (c) CompleteParser	102
Figure 24: Specifying classification details for components	102
Figure 25: Query dialog box	104
Figure 26: The selection dialog after querying the broker	105
Figure 27: The graphical component editor in client panel used to (a) compose components and (b) standalone component	106
Figure 28: The instantiated components with the parameters specified at runtime (a) Preprocessor and (b) CParser	106
Figure 29: Wrapper class for dx2j3d utility	108

Figure 30: The (a) composition and (b) execution of the DXF2J3Dconverter components in the WebCODS environment.....	110
Figure 31: The instantiated components resulting from the composed application (a) file editor and (b) file viewer	111
Figure 32: The tool to manipulate thesaurus (a) loading (b) testing of thesaurus	130
Figure 33: Execution of the broker program.....	131
Figure 34: Sample HTML to execute the client application.....	134
Figure 35: (a) Login screen and (b) environment for the client application.....	135
Figure 36: (a) Querying modes in the client environment and (b) searching components in the broker	136
Figure 37: Listing of matched components from the broker	136
Figure 38: Composition of components in the client environment.....	137
Figure 39: (a) Saving components and (b) the generated description based of the composed application.....	138
Figure 40: Loading of saved components into the environment (a) loading and (b) using the component	139
Figure 41: The resulted error message showing the required components when loading saved description.....	140
Figure 42: Sample HTML to execute the provider application.....	143
Figure 43: (a) Login screen and (b) running environment for the client application	144
Figure 44: Creation of primitive component	146
Figure 45: Specifying details for ports using the create diagram	147
Figure 46: Specifying classification details for components	148
Figure 47: Removing assets from the broker.....	149
Figure 48: Class Diagram for the service package.....	157
Figure 49: Class Diagram for the typing package	164
Figure 50: UML class diagram for the accessdata package.....	172
Figure 51: UML class diagram for the broker package	175
Figure 52: The UML description of the servicecenter package.....	180
Figure 53: The UML description of the searchengine package.....	183

Chapter 1: Introduction

This thesis is about the design and the development of a framework to support brokerage of software components in the Web-based environment. Components referred in the framework can be standalone or composed with other applications. The composition of these components creates flexible applications which can be customized to suit the needs of application users from different domains.

This chapter begins by briefly describing the motivation of this research and a general description of the developed framework. The evolution of the Internet enables the emergence of innovations and electronic markets. The Internet becomes a worldwide resource for programming solutions and provides effective exchange of products between programmers and consumers. The structure can be supported by an electronic intermediate that supports brokerage of components.

The Internet supports the development of software tools that can be downloaded on-demand by users — software tools on-demand. These tools cannot be purchased like products, because they do not reside on the user's machine. Rather, they can be used as "services." The idea is to let users building applications using a wide variety of composable components, provided by several providers who make them available through the Internet. The wide diffusion of Java and JavaScript foster innovative techniques for software distribution. Java-based tools can be downloaded on demand from the developer's server and executed inside a Web-browser; installations of the software are not required on the user's machine. This approach presents several benefits (Yourdon, 1996). Tools are immediately available to any Internet connected computer. Downloaded tools can run on any hardware platform with a Java-capable web browser. Since the tools are downloaded from a central server, users always get the latest version (Gosling and McGilton, 1996). Moreover, since there is no

installation requirement for software, managing a large user base becomes more viable. Maintenance costs for software are significantly reduced (Gupta *et al.*, 1998). The elimination of packaging and physical distribution helps reduce production costs (Hummel, 1996). Succi *et al.* (2000) suggests that the tool on demand distribution mechanism can be enhanced with software pay-per-use or renting mechanism. The usage of software by users is monitored and users will be charged based on their usage of the downloaded software.

The software tools-on-demand mechanism can be extended to software components – *components-on-demand*. This software distribution model can be extended to software components instead of full-featured applications. Each application feature or function in the traditional monolithic application can be decomposed to a candidate tool in the component-based paradigm. Since users do not usually require all the features provided by the monolithic application, software distributors provide a component composition environment allowing users to download and compose their “applications” using only the components that fit their needs. Components users are encouraged to try out more components because of the reduced entry barrier for using new components (Choudhary *et al.*, 1998).

The implication of the component-based paradigm is like the creation of a virtual market. Shaw (1999) analyzes this distribution behavior using the metaphor of the Virtual Agora, a virtual market place where developers and users meet and exchange resources. The creation of the infrastructure requires new toolkits to identify, compose and track the resources available in the agora. Moreover, an economic infrastructure is also required to support the market and charging mechanism. The architecture of the agora is still open since no system successfully supports the operation-model of such a virtual market.

Tools supporting web-based distribution and composition of software components should provide the infrastructure for component-based development. The infrastructure must define a component repository for managing existing components for development. A few tools have been developed, such as ArchStudio (Medvidovic and Taylor, 2000), Resolve (Bucci, *et al.*, 1994; Sitaraman, 1999), and Espresso (Faison, 1997). In most cases, they provide proof of concept for new ideas, focus on specific aspects of component composition, and do not target of the overall requirements of a fully functioning system.

When composing software systems from distributed components that perform robustly regardless of platform or network infrastructure, the use of middleware for the integration of distributed objects is necessary. However, there are still common design problems regarding composition of software components. These problems range from heterogeneity of computing platforms to absence of suitable support tools (Emmerich, 2000). Middleware for connecting distributed objects are available, such as Common Object Request Broker Architecture (CORBA), Component Object Model (COM), Distributed Component Object Model (DCOM), Enterprise JavaBeans (EJB), Java RMI.

The Common Object Request Broker Architecture (CORBA) enables open interconnection objects in an open connection environment. This technology is based on a set of invocation interfaces describing the requirements for invoking the object. The distributed object uses an Object Request Broker (ORB) to locate the receiver object, invoke method, and transport arguments. The message passing process involves sets of object adapters that marshal and unmarshal the arguments, and then invoke the requested method on the receiver object.

The Component Object Model (COM) is the component software framework that Microsoft used on its platform. The fundamental component of COM is the interface.

On the binary level, an interface is represented as a pointer to an interface node. The only specified part of an interface is another pointer held in the first field of the interface node. The second pointer is defined to point to a table of procedure variables (function pointers). The double pointer mechanism allows clients to see only a pointer to a pointer to the virtual functions in other components. In general COM offers an infrastructure for building software applications from blocks of objects.

The Distributed Component Object Model (DCOM) developed by Microsoft extends the COM to support communication among objects on different machines based on client-side proxy objects and the server-side stub objects. This mechanism uses the idea of Object Identifiers (OID) to locate the object in the server and an Interface Pointer Identifier (IPID) to signal the server of the process. DCOM also provides access control to protect the data and the service.

The Enterprise Java Beans (EJB) technology consists of two parts: server and clients. The server is a container for the client to access individual Enterprise JavaBean objects. The server allows the client to identify and create instances of specific Enterprise JavaBeans within itself. The client can then use these objects to perform some function (for instance, business logic). An application may use one or more Enterprise JavaBean servers in its programming. Once the client disconnects from the server, EJB instances that are no longer used by the client are destroyed. The EJB may choose to save the state of variables to secondary storage for later retrieval or discard all information.

The Remote Method Invocation (RMI) is a Java-centric way for distributed computing. It offers the core components needed for any distributed systems in Java-only applications. The object communication scheme of RMI is analogous to CORBA. The objects being transferred across networks are serialized. The

serialization of objects allows users to transfer or request any object by value from one remote process to another. The registry in RMI serves the role of Object Manager and Naming Service for the entire distributed object system. Clients can communicate with the registry to look up objects in the server. Unlike CORBA, the RMI registry is required to be running on the server of a remote object.

Composition of components can be divided into two areas: offsite composition and local composition.

The offsite composition allows component integration to be performed among heterogeneous computing platforms. The integration of resources is performed in the remote site where the components are executed. Standards supporting offsite composition have been discussed above, such as, COM, DCOM, CORBA, and RMI. Modeling and implementing middleware software connectors with complex protocols become a key aspect of architecture-based development. (Dashofy *et al.*, 1999).

However, the increase of computing power in terminals and personal computers encourages the composition of components to be performed in the user's local environment. The composition of components locally allows more flexibility in the connection mechanisms. Currently, there is no standard approach to support local composition of components. Limited works have been reported in compose components using local objects. (Faison, 1997; Weinreich, 1999, Hewlett-Packard, 1999; Michiels and Wydaeghe, 2000)

1.1 Objectives

The primary objective of this thesis is to provide the basis to support a component brokerage system by developing a software framework. The system developed in this thesis is called WebCODS (Web-based Components On Demand System).

WebCODS aims at creating a comprehensive environment where components can be effectively distributed and used through the Web. WebCODS identifies three major roles in the supply-use chain: *Component Providers*, *Customers* and *Brokers*. Providers sell their components to customers through the mediation of a broker. Customers find and obtain the interested components from a component broker. Customers can use the components as they are, or further compose them together in their local environment.

Besides the matching of suppliers and customers in the market place, the intermediary also provides additional services, such as, information of services, categorization of products, trust, and quality assurance. The identification of products in the market requires the development of a standardized production description. The using of unified product descriptions simplifies the evaluation and comparison of alternative products. The information cost for users is significantly lowered in the centralized market space.

The virtual market transfers components and sensitive information via the Internet. The transfer of this secure information in an insecure network needs to fulfill the following security properties:

- Confidentiality ensures the secure information is not disclosed to unauthorized recipients.
- Integrity ensures the data send in the network are only modified or destroyed in a specific and authorized manner.
- Availability ensures that the resources of the system will be usable whenever they are requested by authorized users.

The support provided by WebCODS targets all the key areas for secure component brokerage on the Web and dynamic composition of components. The system is

operated in a Web-based environment, therefore, it allows dynamic uploading and downloading of component to/from providers, customers and brokers. The mobility of components via the Internet is required. New components can be created dynamically by composition of existing components. After composing components in the environment, composers can instantiate and execute the created components. The broker also features a facility for components classification and identification. The management of components is supported in the infrastructure. The system provides security services to protect the overall system operations against unwanted accesses. The infrastructure of WebCODS also contains a security center and search engine. The security center uses public and private key pairs to protect the transfer of components in the Web being tapped by unauthorized users. The search engine handles the identification of components in the broker.

1.2 Thesis Structure

Chapters 2 to 5 discuss the background information related to the thesis. Chapter 2 describes component composition in detail and examines other available component composition environments. Chapter 3 deals with the theory required for classification and identification of components. The security aspects for the component brokerage infrastructure are discussed in Chapter 4. Chapter 5 includes an introduction of JavaSpaces that is used for the primary transfer medium for components in the framework.

From Chapter 6 to 8, the requirements for the WebCODS project are described. Chapter 6 first introduces the architecture of the WebCODS application. Then the following sections target at the aspect of component composition and component brokerage in WebCODS. Chapter 7 is focused on the design strategies to implement the composition environment in the application. Chapter 8 states the requirements for implementation of the component brokerage and the related security issues.

Chapter 9 discusses the implementation details to target the requirements specified in the previous chapters. The discussion is centered on the implementation of the composition environment, the security manager used for transfer components in the network and the search engine for identification of components in the broker. Chapter 10 provides an example to illustrate how WebCODS works. The example describes the steps from building up of components in the broker environment to the execution of the component in the client's location.

Chapter 11 is an evaluative conclusion of the research and includes directions for future work.

Chapter 2: Component Composition

Nierstrasz and Meijler (1995) define software composition as the construction of software application from components that implement abstraction pertaining to a particular problem.” The benefit of composing systems from components is flexibility. Systems created from components should be easy to recompose to address new requirements (Medvidovic *et al.*, 1999). This section describes a few approaches to compose components.

2.1 Scripting Languages

Construction of software applications from existing components has been a major research directive for many years. Sametinger (1997) has identified successful examples of composable software entities existing in many environments, such as mixins, macros, functions, templates, and modules. Scripting languages are designed for “gluing” a set of existing components and plugging them to work together (Ousterhout, 1998). The script is used to specify the composition of program components, and perform late binding of components together. The language for composition can be formalized as Module Interconnection Languages (MILs) and Architectural Description Languages (ADLs).

2.1.1 Module Interconnection Languages

MIL is used to provide a means for the programmers of large systems to express their intent regarding to the overall program structure in a concise, precise and checkable form. It provides formal grammar constructs for deciding the various module interconnection specifications required to assemble a complete software system. Communications in MIL systems are performed by exchanging resources among modules. Resources are entities that can be named in the corresponding implementation language, such as, variables, constants, procedures, and type definitions. Modules are units that provide resources and require some set of resources. Once a system structure is coded in MIL, the system can be verified for completeness and inconsistencies. MILs are coupled with the programming language used to implement the system. (Prieto-Diaz and Neighbors, 1986)

The first module interconnection language, MIL75, was developed in 1970s (DeRemer and Kron, 1977). MIL75 decomposes a large system in a tree representation with modules as nodes. MIL75 has an associated compiler to verify module accessibility and resource management.

Thomas' MIL is based on the idea that module interconnection should be flexible and not constrained in a tree based structure as in MIL75 (Thomas, 1976). The proposed MIL allows software systems to be represented as a "finite directed graph" -- all the contained nodes in the graph are reachable from each other. The idea of subsystems is supported in the MIL. Subsystems are stored in a library structure and can be referred in an MIL program as a module.

Cooprider's MIL expands the basic of previous MILs to include a version control facility and a software construction facility (Cooprider, 1979). The version control ability allows modules to hold different versioning information, so that duplication of modules is prevented. The construction system supports creation of applications from source files, rules and processors. The source file and rules are input to a processor that creates the application.

Conic is an extension of MIL75 to support execution of components in distributed environments (Magee *et al.*, 1989). Conic components are equipped with ports to communicate with the environment. Message passing is the only supported mechanism for component interaction. Runtime component composition is specified by a configuration language and interpreted by an application server that executes components on different CPUs and transfers messages among components. Conic has an associated compiler that supports platform-independence code generation -- the Amsterdam Compiler Kit. The compiler-kit takes in source code and regenerates platform specific binaries for execution (Tanenbaum *et.al*, 1983).

2.1.2 Architectural Description Languages

ADL is a language that provides features for modeling a software systems conceptual architecture, separated from the system's implementation. Comparing to MILs, the language targets coarser-grained architectural elements and their structures. The building blocks of ADL are components, connectors and architectural configuration. ADLs are usually associated with suitable supporting tools, such as, framework generator and system verifier.

Rapide supports modeling and simulation of the dynamic behavior of component-based systems (Luckham and Vera, 1995). Components are defined using interfaces, connections, and constraints. The specifications of connections are embodied in components and connecting mechanisms cannot be changed after deployment. The composition model also limits the communication between components to event propagation. When communication links are instantiated, a specific implementation is required for each connection.

Darwin addresses software architectures of distributed systems (Magee and Kramer, 1995). The language supports dynamic structures. The modeled evolution includes changes in connections between components and the set of component instances. Darwin components are characterized by services -- provide service and contain services. Components are binded together by services. Each connection is represented by a binding between a required service and a provided service. The operational semantics for system specification in Darwin is analyzed with π -calculus (Milner *et. al.*, 1992). The focus of the analysis is based on the binding of services, instead of interaction between components and connectors.

UniCon considers connectors as first class entities (Shaw *et al.*, 1995; Shaw *et al.*, 1996). It supports a predefined set of connectors with defined semantics: pipe, file I/O, procedure calls, data accesses and remote procedure calls. The software architectures are specified in terms of rules that define the admissible connections between connectors and components. The core components are typed, and a type-checking like mechanism is used to ensure the validity of the connection. Components in UniCon can be source code, object-code components, or executable components. The linkage of components requires dynamic generation of glue codes for participating connectors. The creation of code is based on the component descriptions and the source code. The UniCon parser interprets this information and generates C-based Odin files (Clemm, 1998). The Odin files contain the source code and compiler directives for generation of the components recognized in by the Odin

compiler. UniCon is able to deal with executables; it provides limited support for dynamic component composition, but constrained in Unix-style and C-based environments.

Wright is an extension of UniCon that uses formal specification to specify protocols of interaction between components (Allen and Garlan, 1997). The protocols are defined by Communicating Sequential Processes (Hoare, 1985). Since connectors and components are specified with formal specifications, properties of connections can be checked automatically for consistency and compatibility.

The C2 architecture style (Medvidovic *et al.*, 1999) can be summarized as a network of components connected together by connectors for message passing. Components and connectors both have a defined top and bottom. The top of a component can be connected to the bottom of a single connector. The interface port of connectors is typed and it can support an arbitrary of components. The interface of connectors is evolvable and it is determined by interfaces of the connected components. The C2 supports the generation of application frameworks for C++, Ada and Java. C2 also supports dynamic manipulation of the application architecture with the C2'architecture modification language. The language allows add, remove, weld and unweld for components in the architecture at runtime.

ACME is an Architecture Description Interchange Language used to provide intermediate representations for ADLs (Garlan *et al.*, 1997). The core architectural entities supported are components, connectors, systems, ports, roles, representations, and rep-maps. Compositions of entities are interpreted into identified architectural templates. These templates support the abstraction of information from the style and are used to interchange the architecture description between ADLs.

ARMANI provides a design vocabulary, design rules and architecture style to capture details for architectural design (Monroe, 1998). It supports definition of styles in terms of component types, connector types and architectural constraints.

Table 1 summarizes the comparisons of supported features between different MILs and ADLs. Some of them are used only in the analysis of software architecture, such as, Darwin, Rapide, Wright, ACME, and ARMANI. They are not used to relate the software architecture and the actual implementation of the composed system. The table shows that UniCon and C2 is the only ADL that supports dynamic composition of executables. UniCon using predefined connectors to provide more flexibility for connections, while C2 supports only event passing as connection mechanism among components.

ADL	Connectors as 1st Class Entity	Supports Typed Components	Supports Typed Connectors	Supports Dynamic Composition	Works with Execut able
MIL	No	No	No	No	No
Thomas' MIL	No	No	No	No	No
Cooprider' s MIL	No	No	No	No	No
Conic	No	No	No	Yes	No
Rapide	No	Yes	No	Yes	No
Darwin	No	Yes	Yes	No	No
UniCon	Yes	Yes	Yes	Partial	Yes
Wright	Yes	Yes	Yes	No	No
C2	Yes	No	Yes	Yes	Yes
ACME	Yes	Yes	Yes	No	No
ARMANI	Yes	Yes	Yes	No	No

Table 1: Comparisons of supported features of component composition in ADLs

Medvidovic and Taylor (2000) have evaluated a set of ADL based on their properties. They have concluded most ADLs support static generation of source code for the application or the skeleton for the application framework. Clearly, components generated from these ADLs do not support evolution of software architecture during runtime; a certain level support of dynamism is necessary. The paper found that the support tool ArchStudio of C2 exhibits this property. The tool allows interactive construction, execution, and run-time modification of C2 architectures implemented in Java. The dynamism is achieved by loading and linking new components or connectors into the tool dynamically.

2.2 Other Composition Approaches

To connect components together requires multiple steps, most of which require human intervention. Standards for composing components exists, such as, CORBA, COM, DCOM, and RMI. In the last few years, different tool-sets have been developed to allow composers to connect components together.

Rational Rose is a visual modeling tool to improve software design, manage complexity, define software architecture, enable reuse and capture vital business processes (Rational Rose, 2001). The users of the tool are able to create components templates and saved as a model in the Rational Rose standard. The application framework can be composed using the defined component models. The composed framework creates skeleton code and a visual model to the application. The generated application requires experienced technical personnel to fill in the missing parts before the framework develops into an executable application.

Paradigm Plus is an UML-based modeling tool for creating component centric applications (Paradigm Plus, 2001). The focus of the application is in designing, modeling, and component-reuse of software systems. Each component defined in the application is hosted on a multi-user object repository. The application is able to publish the UML-based models of the defined components in HTML format and added to the intranet web site. Therefore, anyone within the organization can browse the design models and locate reusable components. The composition styles of the software components are defined in the UML-based model.

Broadvision's software suite helps companies construct Web sites that offer personalized e-commerce transactions, tailored customer service, and Web-based content management (Broadvision, 2001). The application offers a design center and

a publishing center to Web authors and Internet application developers. The design center provides Broadvision's users a standard approach to create Web-based applications. The publishing center is an easy-to-use tool with a Web-based interface for managing distributed, collaborative online content development. It allows a distributed team of non-technical content experts to manage every aspect of site content, including creation, editing, staging, production, and archiving. The references of the Web pages are linked together statically in the publishing center.

(Kent *et al.*, 1998) proposed a pictorial approach to specify and verify software composition. Using the framework, composition of components is achieved through sub-typing and framework-inclusion. Further information can be found at Catalysis (d'Souza and Wills, 1997). The development environment is a CASE tool similar to UML, which generates skeleton code for component developers. The component composition environment supports a software development methodology, and is not constrained to any OO programming languages.

Ciao is a tool that generates target applications by solving packaging mismatch between C-based components (Deline, 1999). The tool contains a Ciao compiler to process the Ciao code written by the packager and component producer. The target product of the compiler is the packager source and component source in C. The UniCon system is used to generate applications with the available component resources.

RESOLVE aims at reducing the gap between formal specifications and actual implementations through interfaces of components (Bucci, *et al.*, 1994; Sitaraman, 1999). The supporting tool, WorkBench, creates a development environment for RESOLVE language. The tool contains a navigator to browse existing components, a module editor to manipulate specifications and a composition wizard for composing components. Component composition in the tool is based on the matching of formal

specifications for interfaces and implementations of components. The target outputs of the tool are source code of the application either in Ada, C++ or Java, and these codes are passed to corresponding compilers to generate executables.

Composition theory expresses rules for composing components in formal languages. The language is used to specify interfaces of components. Therefore, modules in system can be designed and verified separately. The implementation of the modules must satisfy its interface without knowing details of other modules in the system. The analysis of the interface provides verifications that the collection of interacting modules satisfies the system specification. Examples of composition theory can be found in (Lam and Shankar, 1994), (Fiadeiro and Maibaum, 1995) based on Category Theory (Goguen, 91), (Magee and Kramer, 1995) relied on π -calculus (Milner *et. al*, 1992), and (Allen and Garlan, 1997) using Communicating Sequential Processes (Hoare, 1985).

2.3 Dynamic Component Composition Tools

There are tools available for dynamic component composition for binary components -- Espresso (Faison, 1997), Combo Framework (Weinreich, 1999), E-Speak (Hewlett-Packard, 1999) and Michiels and Wydaeghe (2000). These models are based solely on the interface of components and method invocation. Connections between components are established with direct connection, parameterization of connectors in the source code level is not required. The dynamism of the composition environment allows interactive construction, execution and runtime-modification of the system. To support these behaviors, the framework allows dynamic loading and linking of components into the environment.

Espresso allows users to compose components in a tool-based environment. Espresso components are JavaBeans. The tool contains a user interface representing components as box and line. Users of the tool link components together by drawing

lines across the ports. Ports are not typed, and they refer to methods within the component. Properties of JavaBean also support components to be loaded dynamically and transferable via network into the component environment. Glue code is not required to connect components since the composition model is based on the JavaBean specification.

The Combo component composition framework was developed in C++ and requires components to be loaded at start-up. The framework does not support dynamic clustering of existing components to form new components. Interfaces and properties of components are required to be specified in a separate system configuration file. A component manager uses the configuration file to determine which component is supplied to the application and being executed at runtime. The application determines the required interface and requests the components from the manager. The manager finds the best matching component, then it creates an instance and binds the component with the application. Components in the framework interacts with interfaces, no generation of glue code is necessary.

E-Speak is a component composition framework in a distributed Java environment. The environment takes advantage of the serialization and RMI properties of Java. The environment joins participants in a federation and they are classified as providers and clients. Providers are parties who create instances of components and supplying them via the Web. These components are implemented with a predefined set of interface class in Java. Clients use predefined interfaces to search for matching components within the federation. New components cannot be dynamically loaded into the system without notifying others about the newly predefined interface. Components are supplied as serialized representations (an object-instance for local and a stub for distributed environment) and executed in the target location. Glue code is required to connect component in the framework, so recompilation is required for each newly composed components with the provided interfaces.

Michiels and Wydaeghe (2000) propose a tool that composes components based on micro-architecture design in Java environment. Micro-architectures are templates for different architecture-styles of interactions between components. The environment provides composers with a predefined set of micro-architectures as connectors for components. Components are defined with an application program interface (API) and a set of usage scenarios. The usage scenario also defines the mapping of the API to the interactions in micro-architectures. A component composition wizard is provided in the environment to allow users to specify the behavior of the composed component in the form of message sequence chart. The wizard analyzes the chart and generates the glue code for the micro-architecture template. The code for the template is loaded into the execution environment to execute the composed application. The only supported connection mechanism within the connectors is method invocation.

Table 2 contains a summary of the investigated tools for software composition.

Name	Composition	Summary
Ciao	Static	<ul style="list-style-type: none"> • Constrained in C and Unix Environment • Generate code and compiled into executable
Resolve	Static	<ul style="list-style-type: none"> • Support ADA, Java and C++ • Based on a composition language to describe components • Generate code and compiled into executable
Rational Rose	Static	<ul style="list-style-type: none"> • Based on Rational Rose application framework • Generate skeleton code only
Paradigm Plus	Static	<ul style="list-style-type: none"> • CASE Tool • Support publishing of components • Generate skeleton code only
Broadvision	Static	<ul style="list-style-type: none"> • Web-publishing tool • Allowing dynamic creation of Web page

		<ul style="list-style-type: none"> • Support publishing of components
CORBA	Dynamic	<ul style="list-style-type: none"> • Require predefined interface between components • Support components in different platform and development language
COM	Dynamic	<ul style="list-style-type: none"> • Supports only Window environment • Required predefined interfaces between components
DCOM	Dynamic	<ul style="list-style-type: none"> • Same as COM • Support distributed components
RMI	Dynamic	<ul style="list-style-type: none"> • Support only Java environment • Predefined interface required
Espresso	Dynamic	<ul style="list-style-type: none"> • Based on JavaBean standard • Interface of components do not need to be predefined
Combo	Dynamic	<ul style="list-style-type: none"> • Support C++ environment • Interface of components do not need to be predefined • Interface of components are stored in a database
E-speak	Dynamic	<ul style="list-style-type: none"> • Based on RMI • Require predefined interface between components
Michiels and Wydaeghe	Dynamic	<ul style="list-style-type: none"> • Predefined interface required • Based on micro-architecture templates

Table 2: Investigated software composition tools

2.4 Summary

Nierstrasz and Meijler (1995) suggest the composition of software component divides into two streams: static and dynamic.

The static composition of components creates statically binded applications which can not be altered during runtime. Any changes to the architecture of the application require recompilation and re-execution of the system. On the other hand, dynamic composition allows dynamism in the composed systems. The architecture of these systems can be modified at run-time. The alternation of the system architecture may not require reinstantiation of the application to accommodate the changes in the structure.

Applications created with the static composition are based on compiler and linkers. The composition environment interprets the instructions for composing applications with a parser. The parser generates code for components and glue code for connectors. The generated code is passed to the compiler, which statically binds them together into executables. The component can be executed without the composition environment. Once the system is composed, any changes to the architecture require recompilation of the system. The connection mechanisms among components are not restricted to method invocation and interfaces in component specifications. However, the overhead of code generation and compilation is large.

The dynamic composition technique creates applications dynamically. The technique works with the binary code of components. The composition of components is based on interfaces and predefined communication techniques, such as, method invocation and event passing. Since connections between components are established with direct connection, parameterized connectors are not necessary.

The dynamism of the composition environment allows interactive construction, execution and runtime-modification of the system. Therefore, applications cannot be executed without the composition environment. To support these behaviors, the environment allows dynamic loading and linking of components into the environment. The composed components are not statically bounded together in a rigid

way, and they can be rearranged to form another applications. The target software can evolve at runtime.

The components in WebCODS are binary components; therefore, their composition environment uses dynamic composition. The connection mechanism between components in dynamic composition is limited with component interfaces and method invocations. This limitation is overcome by the use of UniCon. This ADL provides the environment with a composition model and a predefined set of connectors. The set of connectors provides extra connection mechanisms between components in addition to message passing, such as piped streams, file connections, and socket connections. Applications composed in WebCODS allow dynamic alteration of system architecture at runtime.

Chapter 3: Component Classification

Background

Building a software component library requires a classification method to organize the contained assets. To successfully identify and reuse the components in systems, users must be able to understand the classification scheme and find the desired assets. Prieto-Diaz (1991) says that a classification scheme for reusable software must meet the following criteria:

1. Support an expanding collection
2. Locate exact matched and similar components
3. Find functionally equivalent components across domains
4. Detail components precisely and descriptively
5. Be easy to maintain and use
6. Be amenable to automation

3.1 Representation of Components

Frakes and Pole (1994) describe an empirical investigation for the different libraries and information science methods. The representation of information can be divided into two different categories: controlled vocabulary and uncontrolled vocabulary.

Controlled vocabulary places limits on the terms that can be used to describe classified object and on the syntax that can be used to combine those terms. Controlled vocabularies rely on a predefined set of keywords as indexing terms derived by experts. The controlled vocabulary thesaurus controls all acceptable terms that can be used and unacceptable terms that are not usable. Uncontrolled vocabularies do not have restrictions on the terms and syntax in the component descriptions.

The representation of components can be decomposed into enumerated, facets, attribute-value, and free text (uncontrolled) keywords classifications.

Enumerated classification divides subject areas into mutually exclusive hierarchical classes (Booch, 1987). The highly structured schema makes enumerated classification easy to understand and use. The well-defined hierarchy helps users understand the relationships among controlled indexing terms, and provides a searching method by traversing the tree. For example, the structure of the Java Foundation Class (JFC) library is classified into different packages according to their functionality. However this classification requires a lot of domain knowledge to create exclusive relations. Once the classification scheme is established, a simple change of the scheme may affect other assets in the scheme. Therefore, this classification scheme cannot be used when the domain is not well understood and evolving.

Facet-based classification divides subject areas using basic terms that are organized as facets. (Prieto-Diaz, 1985; Prieto-Diaz and Freeman, 1987) The facets are identified by the most important vocabulary in a domain and then grouping the similar terms together into facets. Assets are classified by synthesizing the facet term pairs in the classification scheme. Facets are more flexible than enumerated schemes because individual facets can be redesigned without impacting other facets. This method requires that users know the significance of each facet and the terms that are

used in the facet. The list of facets can be ranked according to their importance, and usually the size of the list contains less than seven items. The schema can be enhanced with a controlled vocabulary thesaurus to provide the searching ability across domains. Facet-based classification also provides some facilities for handling synonyms that are not available in attribute-value matching.

Attribute-value classification describes information in terms of a set of paired attributes and values. This classification scheme is very similar to facet classification. However, the vocabulary used can be controlled and uncontrolled. The faceted technique uses a fixed number of facets per domain, and no such restriction exists for attribute-value methods. Moreover, searching of attributes contains no order and the list of attributes is not constrained to any numbers of items.

Free-text keyword indexing uses terms automatically extracted from documentation. (Frakes and Negmeh, 1987) The free-text analysis consists of analyzing word frequencies in the natural text. Relevant keywords are derived automatically by their statistical and positional properties. The processing is automatic and is referred to as automatic indexing. The extracted terms in free-text analysis can be drawn from any source. Since the indexing is automated, this results in lower maintaining cost than other classification schemes. The terms used in classification is not restricted, they can be as specific as possible to describe assets. The low cost of building the repository coupled with adequate performance has made this approach popular in commercial text retrieval systems and World Wide Web search engines such as Yahoo and AltaVista.

The study of Frakes and Pole (1994) also suggest that no single indexing method works really well. The effectiveness of these systems ranged from 40% to 60%. The evaluation is based on the recall and precision number. Recall is the number of relevant items retrieved over the number of relevant items in the database. Precision

is the number of relevant items retrieved over the number of all items retrieved. Moreover, different representation methods will locate different items, although their recall and precision numbers tends to be similar. Users tend to use systems that are easy to use and understand.

Information retrieval systems need to address incomplete and inconsistent indexing. The downfall of the traditional matching strategy is that queries are viewed as precise specifications of user needs and document representations as precise descriptions of repository objects. The existence of incomplete and inconsistent indexing makes the retrieval methods and algorithm go beyond simplistic keyword matching schemes. The employment of “intelligent” retrieval methods extends the exact-match paradigm to retrieve items by reformulation of the query (Williams, 1984). The reformulation can be performed by combining different representation models to obtain a higher accuracy and flexibility in classifications and retrieval of components. Empirical studies indicate that this combination of techniques achieves good performance in the face of indexing problems and ill-defined information needs (Henninger, 1997).

Table 4 summaries the characteristics of different classification methods.

Classification	Features
Enumerated	Rigid structure, difficult to change, requires concrete domain knowledge
Facet-based	Flexible structure, easy to change, requires understanding from users, facets are ordered
Attribute-value	Flexible structure, attributes are non-ordered
Free-text	Non-structured, using natural language for description

Table 3: Summary of classification scheme

3.2 *Software Libraries*

(Creech *et al.*,1991) proposed a tool called Kiosk, to interconnect a library of reusable assets using hypertext. The hypertext is selected for structuring the library because it supports navigation while performing searches within the library. The use of keywords and free-text searching are also supported by the hypertext. The software components in the library are Unix text files with links that point to other nodes in the library. The nodes are classified into two forms: classification nodes and component nodes. The classification nodes provide the classification lattices of all its referred nodes. The referred nodes can be other classification or component nodes. Component nodes contain the actual functionality of the referred component. The library structure is highly flexible, because an existing library can be restructured by linking nodes in different arrangement. However, the experimental users of the system claim that it is difficult to use and that is easy to get lost in the hypertext environment.

Henninger (1997) developed a software reuse repository for Emacs Lisp functions and variables. The repository supports tools to create information structures, flexible mechanisms to search and browse the repository, and tools to refine and adapt information as users work with the repository. The components are extracted from text files and indexed with a combination of automatic extraction and interactive user support. Components in the repository are indexed with key terms and phrases. The system uses an intelligent retrieval method that extends the exact-match paradigm to retrieve items that are associated with a query. The method uses an association network for terms. The network consists of two layers of nodes: terms and components. They are connected with weighted links. The retrieval of components is based on a spreading activation procedure

(Mozer, 1984) applied on the association network. The query node is given an activation index. If a node has an index greater than a threshold, the value is passed through each of its links to other nodes. The sum of the received activation value is modulated with a decay parameter -- the parameter is calculated based on the weight

of links. With this technique, an unlimited number of documents may be “relevant” to some degrees. The stabilization of the matching set is achieved when the maximum number of iterations is reached, or the activation index lies below certain threshold. The association network adapts to the evolving environment by adjusting the linkage weight between nodes according to the feedbacks from users.

Swanson and Samadzadeh (1992) implemented a software library that uses facet-based classification. The system is used to catalog software components using the same faceted cataloging scheme presented by Prieto-Diaz and Freeman (1987) and supports retrieval of components from the repository. The system is used to only catalog components at the source code level. Six facets used are divided into two major areas: 3 for functionality of the components and 3 for describing the execution environment. Each facet used is accompanied with its own thesaurus data file. The thesaurus is filled with sets of descriptors. For example, given a set of descriptors “add, sum, total” for the facet “function”, the user can use any of the descriptor as input for classification. If the user enters the word “total”, the thesaurus searches the set and understands the input term referred is the same as “sum” and “add”. The system does not support dynamic insertion of terms into the thesaurus. The retrieval of components is based on matching of facets. The studies also suggests more specialized the components, they are easier to be classified by facets.

Jade Bird Component Library system (JBCL) is a library system which supports classification, organization, storage and retrieval of components (Li *et al.*, 1998). Components in the library are classified with attribute-values, keywords, enumerations, and facets. Component providers can also supply information about the relationship between the provided component and others in the library. The system uses facets as the core classification method, and the other classifications are auxiliary. The following five characteristics of components are selected as facets: Application Environment, Application Domain, Functionality, Level of Abstraction, and Representation. These facets are orthogonal and independent of each other, and

reflecting sufficient characteristics of components. Each facet is associated with a structured set of legal terms called a term space. The component providers are not constrained to the existing terms. They can add in new terms to the term space and associate the term with synonyms. Users can retrieve components by facets, keywords or attributes. After a component is found, the users can navigate the relations based on the supplied relationship information by the component provider.

(Damiani *et al.*, 1999) presented a hierarchical-aware classification schema for object-oriented code. The components in the library are classified with a faceted Software Description (SD). The SDs are generated automatically by extracting *verb*, *noun* pairs from interfaces of the source. (Biggerstaff *et al.*, 1994; Etzkorn and Davis, 1997) These pairs form the features of the components and classify the software components according to their behavioral characteristics. Each feature is a SD associated with a specific weight used to compare relevance with other SDs. The assignments of weights are performed automatically with a fuzzy weights assignment technique. Since the system is designed for object-oriented code, the inheritance properties of OO programming language is also presented in SDs. The propagation of contexts in the SDs results in a hierarchy of component library with a classification tree used to select contexts. Although a large portion of the classification work is done automatically, the system still relies heavily on an application engineer who customizes the automation script and manual tuning for each single SD. The retrieval system requires the user to set up a features list of the desired component and their weights used in searching.

Component Behavior Library (CBR) classifies component based on the history of executed program traces as a knowledge base to describe the behavior of components (Pozewaunig and Reithmayer, 1999). The assets in the library are required to test carefully. The classification details include the test cases that used to sample the component and “past pictures” of the program. The identification process is based on a signature matching system. The library is divided into partitions containing

components conformed to a generalized signature. The library also contains a runtime environment to support testing and identification of components.

CHIME uses the browser to provide Web-based source code browsing. The central focus of CHIME is the task of inserting HTML links into the source code. Specifically, the library focused on distributed repository, and that syntactic and semantic inter-relations between are preprocessed by some analysis tools. The links in the assets are inserted dynamically according to the request of the searcher based on the CHIME language. The mechanisms for identification of asset are defined using CGI gateway tool. The using of CHIME provides a flexible library structure for the librarian, and searching mechanism for users.

Table 4 shows the summary feature sheet for the described libraries.

Library	Assets	Classification
Kiosk	Source Code	Keywords
Swanson and Samadzadeh, 1992	Source Code	Facet-based
Henninger, 1997	LISP functions	Attribute-value with weighted relations
JBCL	Source Code and Executable	Facets-based and free-text description
Damiani <i>et al.</i> , 1999	Software Descriptors	Facets-based and Attribute-value
CBR	Software Components	Behavior and test cases
CHIME	Source Code, Text Documents	Facet-based

Table 4: Software libraries and their classification scheme

3.3 Summary

Structured repositories for software reuse are faced with two interrelated problems: (1) requiring the knowledge to initially construct the repository and (2) modifying the repository to meet the evolving and dynamic needs for software development organizations. The structure of a repository is the key to obtaining good retrieval results. No matter how “intelligent” the matching algorithm, if the components are indexed or structured poorly, it will be difficult to achieve good retrieval performance. (Henninger, 1997) Moreover, the structuring of software repository methods relies heavily on classification, which requires costly reclassification and domain analysis efforts before a repository can be used effectively. (Frakes and Poles, 1994)

Simplicity is another key for successful repositories. The strength of methods that use sophisticated information structures is that the knowledge contained in the structure lead the users to relevant information easily. The weakness is that no support is possible if the information is not structured in the manner expected with users (Halasz, 1988). However, the good structure is crucial in a retrieval system’s effectiveness, a balance must be found between structure and effective retrieval. Retrieval methods employ very little structure requirement yet yield effective retrieval performance. Feedback can be obtained from users and used to improve the existing structure. (Damiani *et al.*, 1999) (Henninger 1997) The result is a flexible integration of retrieval and repository construction methods that improve the underlying structures as the repository is used.

Chapter 4: Network Security Background

The widespread growth and the public access nature of the Internet enhance the importance of “security”. Downloading and executing code from anywhere on the Internet brings security problems along with it. Oaks (1998) suggests that a secure system contains the following features:

- Safe from malevolent programs: Programs should not be allowed to harm a user’s computer environment.
- Non-intrusive: Programs should be prevented from discovering private information on the host computer’s network.
- Authenticated: The identity of the parties involved in the program should be verified.
- Encrypted: Data that the program sends and receives should be encrypted.
- Audited: Potentially sensitive operations should always be logged.
- Verified: Rules of operation should be set and verified.
- Well-behaved: Programs should be prevented from consuming too many system resources.

In fact, all these features could be part of a secure system, but the need of each of these features varies from application to application. For example, the security model of Java uses the idea of a sandbox to handle the first and second features in the list. The sandbox provides an environment where the program can be executed. The boundary of the environment is variable based on the number of system resources which need to be protected. The Java security model can be extended to handle authorization and encryption. Encryption and digital signatures can be used to protect messages and verify the originator of the message using keys and certificates.

4.1 Keys and Certificates

Keys are required to create and verify digital signatures. Generally, a key is a long string of numbers with a special mathematical property. The properties of keys are based on the cryptographic algorithm they are going to be used for. Keys can either come alone as a symmetrical secret key or in pairs of asymmetrical public and private key. So there are three types of keys – secret, public and private. Under a cryptographic perspective, there are only two types of keys – secret and shared. The secret and private are classified as secret keys – the owner of the key must keep them safely. The public keys can be published in the network, so they are classified as shared keys. The usage of these keys is described here below.

The algorithm using the symmetrical secret key requires all involved parties to use the same key. All parties must agree to keep the key secret, otherwise the security connection between the parties is broken. The problem with this approach is the storage of the keys. It is very important to keep the key secret, since anyone with the key can decrypt the data. Therefore, sending the key over the network without encryption is dangerous; doing so would be the same as sending the data itself unencrypted.

Using key pairs allows keys to be transferable over the network. The public key can be published to the world, as long as the private is kept secret. Encryption can be done with either public or private key. If the information is encrypted with a private key, the decryption must be done with the corresponding public key. The key pairs provide asymmetric operations to cryptographic engines.

When keys are published to the Internet, it does not provide any information about the owner of the key. For example, there is no way to verify that the received key of person A is the key from person A. Since the key is only a set of numbers, the key can be generated by different identities who claim to be person A. There is no way to verify the source of the key based only on the key itself.

The use of certificates solves this problem by verifying public keys with a third-party organization called a Certificate Authority (CA). A certificate contains three pieces of information:

1. The information about whom the certificate has been issued for.
2. The public key associated with the owner.
3. A digital signature that verifies the information of the certificate. The certificate is signed by the issuer of the certificate.

The certificate provides assurances that the public key contained in the certificate does indeed belong to the entity that the certificate authority says it does. The certificate is verified with the digital signature of the CA. If the signature is valid, the public key in the certificate does in fact belong to the entity the certificate claims. The certificate is the primary object for a person to provide other people with his/her public key.

The most common certificate format is X509. An X509 certificate has a number of special properties extending the basic certificate. The certificate is valid only for a certain period of time specified by a start and an end date. Each certificate issued by a certificate authority has a unique serial number. The combination of the serial number and the certificate authority guarantees a unique certificate.

4.2 Encryption Technique

Message digests is a method to ensure that a message passed to and received from the network has not been modified. A message digest is a small sequence of bytes that is produced when a given set of data is passed through the cryptographic engine. The algorithm does not operate with any key. Instead, it takes a stream of data and produces a single output. The message digest does not produce a very high level of security because there is nothing to prevent someone from changing both the text and the digest stored in the file such that the new digest reflects the altered text. A modified version of the message digest is the Message Authentication Code (MAC). The algorithm concatenates a secret passphrase at the end of the data stream and calculates the new message digest. The message digest is considered equal if the same data and the passphrase are used in both the message sender and receiver. The algorithm strongly requires that the sender and receiver agree on what data to use for the passphrase and the passphrase cannot be passed along with the text. Therefore, the security of the algorithm depends heavily on the security of the passphrase.

There are two different types of encryption algorithms available: Secret key algorithm and Public key algorithm.

The secret key algorithm uses a key that is agreed upon by the sender and receiver. The target message is encrypted with the secret key. The encrypted message can be decrypted with the same secret key used for encryption.

The public key algorithm uses shared keys: a private key and a public key. The keys used to encrypt and decrypt are not symmetrical: the private key is used to encrypt the message while the public key is used for decryption. The party in possession of the public key may read the message, but cannot modify and decrypt the new message with the same key again. The key can also be used as a digital signature. The owner of the information signs the document with the private key, and the recipient of the information uses the public key of the owner to verify the digital signature. The signature process contains two steps. The signature engine calculates the message digest for the input data, and then the digest is encrypted with the private key. The verification of the digital signature is the reverse process of signing. The message digest of the input data is calculated, then this digest is passed to the cryptographic engine for decryption with the public key of the signer. The final step is to ensure the decrypted digest and the digest from the data is the same.

Symmetrical keys can also be generated with key agreement algorithms by sharing some public information. Diffie-Hellman Key Agreement (Diffie and Hellman, 1976) uses public share keys to generate the same secret key while preventing parties unrelated to the agreement from generating the same key. In order to encrypt the message, the common secret key is generated from the private key of the sender and public keys from all receivers. The message is then encrypted with the created secret key. For receivers of the message, the common secret key is generated using the corresponding private key and public keys from all participants. The shared information is the public keys of all entities who have potential access to the decrypted message. The agreement ensures that the shared information is not enough for eavesdroppers of the conversation to calculate the same shared key. Boneh and Venkatesan (1996) suggest that it is possible to crack the algorithm only by knowing the most significant bit (MSB) in the Diffie-Hellman Key Agreement Protocol. However, the task is as hard as computing the secret key generated by the algorithm (Bihem *et al.*, 1999).

Oaks (1998) suggests that the use of signed message in Diffie-Hellman Key Agreement can also avoid the risk of “man in the middle attack”. For example, PartyA and PartyB want to build a common secret key and PartyC wants to crack the code. The man in the middle attack is easy if PartyC can impersonate both PartyA and PartyB. If PartyC can answer to PartyA as PartyB and the same on PartyB side, it is easy for PartyC to know the two secret keys and make a bridge between PartyA and PartyB. PartyC can read and modify the exchanged message at the same time. If PartyA and PartyB signed the message, it is impossible for PartyC to impersonate them.

4.3 Summary

The development of a network sharing applications requires the creation of suitable security mechanism to control access for authorized users. It is necessary to provide authentication and encryption services for secure applications.

Daily computer applications using public and private keys can be found. For example, email systems provide services for signing and encrypting messages. These systems provide a key management system for users to store certificates from other entities and the private keys of themselves. The Web-browsers also provides abilities to manipulate certificates and digital signatures. Kemmerer (1997) claims that these features are not widely used at that time. However, certificates and digital signatures are commonly used in Web pages. With the increasing use of Java applets, the Web-browser is capable of downloading and executing programs from anywhere on the Internet. Security precaution is required to prevent breaches from running code from untrusted sources.

Java 1.1 supports the properties of authentication with the introduction of a jarsigner utility. The architecture of Java provides a key management system for managing key and certificates. The jarsigner tool obtains the private key of the entity that is signing the jar file. Once the signed Jar file is produced, it can be downloaded by

users. In order to verify the signature, the certificate of the signer must be imported to readers' key management systems in advance. When the Jar file is read on the remote system, the key management system is consulted to retrieve the public key of the entity that signed the Jar file, so that the jar file signature can be verified. This approach is commonly used in the Applet-viewer and other Java-enabled browsers for web pages containing Java applets (Scott, 1999).

GIShape is a network-distributed system that uses certificates for security (Puliafito *et al.*, 1997). The aim of the application is to provide user access to the Sharpe tool (Symbolic Hierarchical Automated Reliability/Performance Evaluator) through the Web. The Web-page interfacing the application provides an Applet for users to access the application. The secure communication protocol between the application server and clients is divided into 3 stages: registration, initialization and data transfer. The registration stage requires the client to generate a set of key pairs. The client keeps the private key safe and sends the public key to the server. The server creates a certificate for the client's public key and signs it with its own private key. The server then delivers its own certificates to the client and the new certificate issued. The initialization stage is the loading of the Applet in the Web-browser. The client sends a code to the server to identify its public key and a signed message for verification. After the connection protocol is established, secure messages can be sent. The message sent in the data transfer stage contains the encrypted data and the sender's signature. The message is encrypted with the server's public key and signed with the client's private key. The key and certificate operations are fully supported by Java architecture. These operations are managed by the Applet running in the client side and a wrapper for applications encoded in Java executing in the remote site.

The digital signature and encryption algorithms are the tools for application developers to develop secure network applications. Different combinations of using these tools provide flexibility for the developers to create secure applications that

target different security needs. The issue targeted by WebCODS is to develop a secure protocol to transfer components in the network.

Chapter 5: JavaSpaces

JavaSpaces is a distributed object system developed by Sun (Sun Microsystems, 1998). It focuses on transactional Linda operations and is intended as a platform to provide distributed persistence for exchangeable objects. Linda defines an abstraction for programming software agents and defines a small set of coordination operations (Gelernter, 1985). In a Linda-based system, an ensemble of agents work together on tasks within a shared environment, called a tuplespace. A tuplespace contains tuples, which are structured containers of information relevant for the application. The primitive operations provided by the system are retrieving, writing and matching tuples in the tuplespace. (Sun Microsystems, 1998; Ciancarine *et al.*, 1998)

5.1 Overview

The distributed application paradigm supported by JavaSpaces is based on remote agents interacting with others indirectly through a shared data object space. A space is a shared, network-accessible repository for objects. Processes use the repository as a persistent object storage and exchange mechanism. The processes coordinate by exchanging objects through spaces, instead of communicating directly. The space provides a unique set of key features for programming network applications.

Spaces are network-accessible “shared memories” that many remote processes can interact with them concurrently. The “shared memory” also allows multiple processes to simultaneously build and access distributed data structure, using objects as building blocks.

Spaces provide reliable storage for objects. Once the object is stored in the space, it will remain there until a process explicitly removes it. Process can specify a “lease” time for the existence of the object in the space, after which it will be automatically destroyed and removed from the space. When objects in a space are persistent, they may outlive the processes that created them, remaining in the space even after the process has terminated. This property is significant and necessary for supporting uncoupled protocols between processes.

Objects in a space are identified by an associative lookup, rather than by memory location or identifier. The associative lookup provides a relatively simple mechanism of finding objects according to their content. The identification of an object requires a matching template. The template is an object with some or all of its fields set to specific values used for matching processes. The object in the space matches a template if it matches the template’s specified field exactly.

JavaSpaces technology provides a transaction model that ensures that operations on space are atomic. A transaction is atomic in the sense that all or none of the operations in the transaction will be carried out. If any of the operations within the transaction fails, then the entire transaction fails and any sub-operations that had succeeded are “rolled back,” and the space is left in the same state it would have been in if the transaction had never been attempted. Transactions are important when dealing with partial failure.

The space also allows processes to exchange executable content. While in the space, objects are passive data. They are not subjected to modification and execution. However, when objects are retrieved out of the space, a local copy of the object is created for the process using the object. The local object is just a regular Java object. The process can modify the public fields and invoke the methods presented in the

downloaded object. This capability gives a powerful mechanism for extending the behavior of applications using JavaSpaces.

5.2 Operations

There are four kinds of primary operations for JavaSpaces. The operations are:

1. write – write the given entry into the space
2. read – read an entry that matches the given template from the space
3. take – read an entry that matches the given template from the space and remove it from the space
4. notify – notify a specified object when entries that match the given template exist or are introduced into the space

Figure 1 and Figure 2 shows a simple HelloWorld example using JavaSpaces architecture. The sample code contains an Entry that stores the message “Hello World”. The main program uses the write and take function from the JavaSpaces to transfer the entry between the server and the client’s environment.

```
public class HelloEntry implements
net.jini.core.entry.Entry {

    //The message contained in the entry
    public String message
        = new String("Hello world");

    //Print out the message
    public String toString() {
        return message;
    }
}
```

Figure 1: Sample Entry from the HelloWorld example

```

public class HelloWorldExample {
    JavaSpaces jp;

    . . .

    //Write the entry to the space
    public void writeToSpace() {
        Entry e = new HelloWorldEntry();
        jp.write(e);
    }

    //Read the entry from the space
    public Entry readFromSpace() {
        Entry e = new HelloWorldEntry();
        //Set the content to null for matching
        e.message = null;
        return jp.read(e);
    }

    //Main function
    public static void main(String[] args) {
        //Set up reference to space
        getJavaSpaces();

        writeToSpace();

        Entry matchedEntry;
        matchedEntry = readFromSpace();

        //Print out the contain of the Entry
        System.out.println(matchedEntry);
    }
}

```

Figure 2: Sample code for writing and reading the HelloEntry from the space

Entries are the common base-object of all space-based applications. The entries are exchanged in the space to enable process communication, synchronization and coordination of activities. When an entry is added to a JavaSpaces, the entry is stored in serialized form by independently serializing each field in the entry.

A write places a copy of an entry into the space. The entry passed to the write is not affected by the operation. Each write operation places a new entry into the space, even if the same entry object is used in more than one write.

A read operation looks up the space for any entry that matches the template provided as an entry. If a match is found, a copy of the matching entry is returned. If no match is found, a null value is returned.

The matching of entries is based on the following rules:

1. The template's type is the same as the entry's, or is a supertype of the entry.
2. Every field in the template matches its corresponding field in the entry:
 - If a template's field is specified to be a wildcard (null value), then the entry's field is matched.
 - If a template's field is specified to contain a value, then it matches the entry's corresponding field of the two have the same value.

Successive read requests with the same template on the same space might or might not return the same object, even if no intervening modifications have been made to the space. Each invocation of read returns a new object, even if the same entry is matched in the space.

A take request performs exactly like the corresponding read requests, except that the matching entry is removed from the space. Two take operations will never return copies of the same entry. If two equivalent entries are in the space, the two take operations will return equivalent entries but different copies of entries.

JavaSpaces provides services for delivering notifications among processes. The service is based on the observer design pattern (Gamma, 1994). The model works together with event sources, event listeners and event objects. An event source is any object that "fires" an event, and an event listener is an object that listens for fired events. Whenever the event source fires an event, it notifies each of its registered listeners by calling a method on the listener object and passing it an event object. Space-based distributed events are built on top of the Jini Distributed Event model.

The model extends the Java event model to allow events to be passed from event sources in one Java Virtual Machine (JVM) to remote event listeners residing in another JVM.

When using space-based distributed events, the space is an event source that fires events. Events are fired when entries are written into the space that matches the template specified by the event listener. The `JavaSpaces` interface provides a `notify` method that allows processes to register an object's interest in the arrival of entries that match a specific template. When an entry arrives that matches the template, an event is generated by the space and sent to the registered object in the form of a remote event object, by calling a `notify` method on the listener.

5.3 Benefits

Space-based communication loosens the ties between senders and receivers of information. It promotes a loosely coupled communication style in which senders and receivers of information interact indirectly through a space. In fact, the communication protocol in the space behaves like the convention "Message Passing": one process sends out a message, and another process receives it. However, the conventional message passing supplied by most low-level communication forces senders and receivers to know each other's identity, location and existence at the same time. All three must be specified explicitly by the process in order for a message to be delivered. The requirement tightly couples the sender and receivers together.

The loosely coupled communication provides several advantages for communication (Freeman *et al.*, 1999). Senders and receivers can communicate anonymously. Senders of information do not care who gets it, but some process will eventually know how to handle the message. On the other hand, the process that picks up the message does not care about who sent the message. Senders and receivers can be

located anywhere in the network, as long as they have access to an agreed-upon space for exchanging messages. Even if the sender and receiver roam from machine to machine, the communicate code used in the program does not need to be altered. With space-based communication, senders and receivers are able to communicate any anytime. The persistence of messages in the space allows space-based applications to exhibit the “time-uncoupled” property.

Chapter 6: WebCODS

WebCODS is a prototype system to illustrate the idea of components-on-demand in Web-based environments. WebCODS is a system to support brokerage of software components on the Web. The system contains tools to support component composition, search and identification of components in the component broker, and secure transfer of components in the network. Three major kinds of actors participate in WebCODS: *Providers*, *Brokers*, and *Customers* (Figure 3).

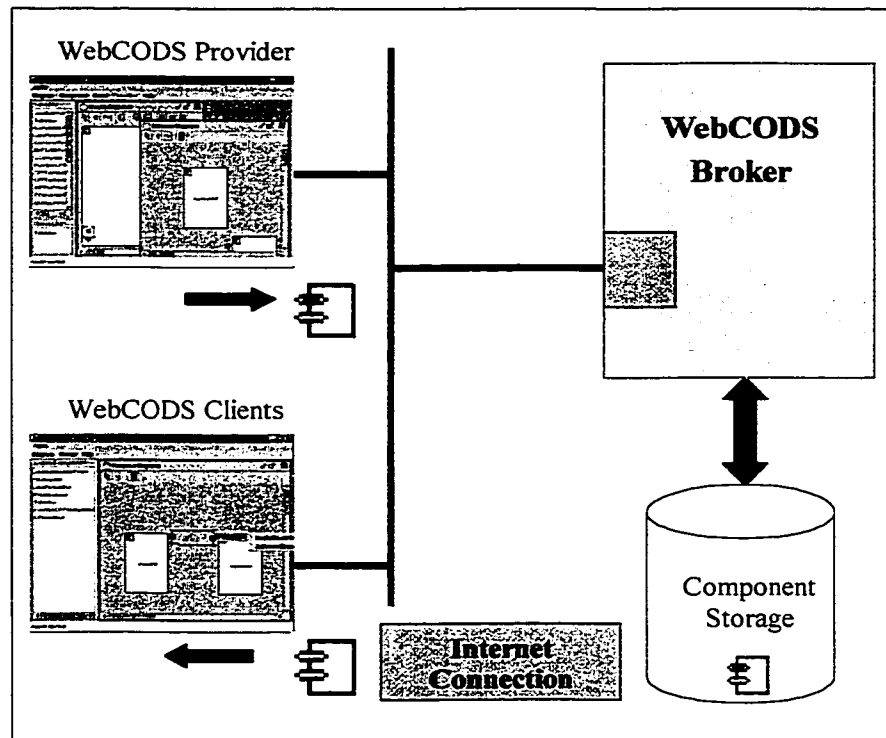


Figure 3: Structure of WebCODS

The Broker provides secure component brokerage in the Web-based environment. It provides an environment similar to a market where Customers and Providers meet and trade products. Customers enter the market to browse and retrieve software

components that fit their needs. Customers who are interested in a component can search the broker for matches or pick from a list containing all available components. The broker transfers the required product to the client's environment for composition and execution. Providers develop software components and advertise their software products through the mediation of the broker.

6.1 *Client Environment*

The client environment provides a login procedure for connecting clients to the broker. If the client is verified as an authorized user in WebCODS, they can connect to the broker and retrieve components from it. The environment contains a service manager to communicate with the broker and a composition manager for managing component composition and execution. The environment also contains a security manager for secure communication between clients and the broker.

The service manager provides facilities for users to query the broker. The querying results in a list of components, which are currently supplied by the broker. The result list of the query contains components' classification details and their providers' information. The manager also handles the downloading of components. If clients want to use a component listed in the broker, the manager sends the request to the broker. The broker processes the request and returns the requested component to the clients using the secure manager.

The composition manager provides the ability to compose and execute the downloaded components in the environment. WebCODS specifies interfaces and connections of components using a notation based on UniCon (Shaw *et al.*, 1996). Composition of components is supported dynamically using UniCon-based connectors with a component composition GUI. WebCODS is based on UniCon, thus the self-standing entities are components, connectors, and ports. The only components supported by the current version of the system are Java components and

their compositions. The system is developed in Java to take advantage of reflection, dynamic binding, and secure class loading.

The security manager provides secure communication between clients and the broker using keys and certificates. The manager uses JavaSpaces as the primary transfer medium. Digital signing and encryption are used in together with JavaSpaces when transferring components in the network. Digital signing allows the receiver to verify the source of information, while encryption ensures that only the authorized parties are able to read the messages.

6.2 Provider Environment

The provider environment is similar to the client environment. It also possesses a composition manager and a security manager. The major differences between the provider and client environment is the ability to upload components to the broker and specify classification details for created components. These extended abilities are handled by the component manager.

The component manager allows users to send the selected components stored in the local component repository. Providers can manage their component list in the broker by adding and removing components from the broker. The broker maintains a dynamic list of available components supplied by all WebCODS providers. The manager also provides facilities for users to classify their components. The classifying information is being used by the broker for searching, and by clients for understanding and identification of components. The classification scheme is based on facets and free-text description.

6.3 Broker Environment

The broker contains a storage area to temporary save the submitted components from providers. Each component is accompanied with the corresponding classification details. The identification of components in the broker requires a search engine. The

engine uses the query prepared by searchers and matches the query based on the facets and free-text description.

The broker provides two ways for clients to retrieve components:

- List all components available in the broker
- Query for a list of available components that fits the searching criteria

The search engine is accompanied with a thesaurus. The thesaurus specifies all the synonymous and ranks them according to the closeness to each other. The query can be expanded using the thesaurus. Therefore, the search engine is able to generate the biggest result set, with ranking of component based on the fitness to the searching criteria.

Chapter 7: Support for Dynamic Composition

The focus of WebCODS is to provide component brokerage in a Web-based environment. The component broker is the administrator of the market; it provides a framework for providers to advertise their software products. Customers enter the market to browse and retrieve software components that fit their needs. The client's environment is responsible for composition and execution of the downloaded product received from the broker.

(Nierstrasz and Meijler, 1995) claim that tools supporting composition of software components should supply three levels of support:

1. Framework Level
2. Composition Level
3. Instance Level

The composition framework in WebCODS allows dynamism and expands the connection mechanism with connectors. In this section, we discuss how WebCODS is targeting these levels and needs.

7.1 Framework Level

The framework level describes the ways components are composed together in terms of component interfaces, composition mechanisms, and composition rules. WebCODS is based on an ADL description called UniCon to express the software architecture in terms of component interfaces, composition mechanisms and composition rules.

Software architectures are specified with ports, connectors and components. Ports are used to define interfaces of components and connectors are used to link ports together. Components are referred to functional programming parts of a monolithic application. Each component has an associated list of ports that defines the supported mechanisms for its interconnections. A set of connectors is predefined in the environment to provide composition mechanisms and to enforce the validity and rules.

Figure 4 shows a sample interaction between components, connectors and ports. There are 2 components (A and B), 4 ports (ζ and ω , ports of A, and ψ and ξ , ports of B) and a connector (x). Port ζ of component A is connected to port ξ of component B using connector x .

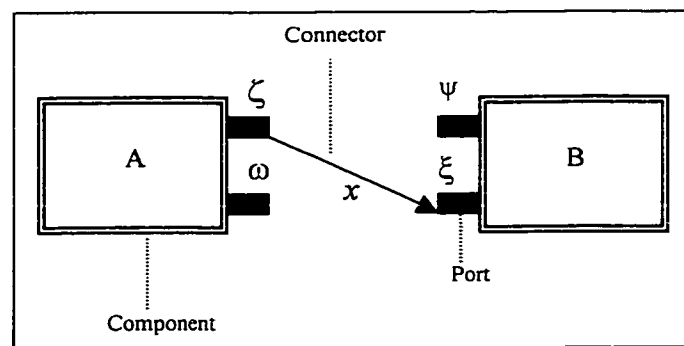


Figure 4: Software Components and Connectors

Components supported are primitive and composite. The basic building blocks are “Primitive Components”. They refer to primary features of applications. “Composite Components” are defined as a collection of other (primitive or composite) components. These components are interconnected together to form complex features required by the composer.

Components in the framework are transferred between broker, clients, and providers via networks. The transfer of the component requires the full specification of components, which contains two parts: (1) the description of internal architecture of the composite components and (2) the executables required to run the application. Therefore, a protocol must be defined for the transfer of this information.

7.2 Composition Level

Composition level defines how applications are composed with components within the framework. Specific applications are obtained as compositions of other generic components defined in the framework. These newly created applications can feed back to the framework as new components. The composers performing components composition need to be aware of the connectivity of the components and perform composition. The details regarding the implementations of components are not required.

Garlen and Shaw (1994) performed a series of work in the area of Software Architecture. During their studies of industrial projects, they have identified a few common architecture styles that are usually used, such as, client server, and blackboard architecture. In order to formalize a way to perform checks and analysis in different architecture style, they decided to break down the system into a set of components and connectors. Therefore, each component is specified by a typing description of itself for function classification and a set of players which interacts

with the outside world. The connector specifies rules for components in the connections, such as, number of connections and algorithm for connection.

The analysis of Software Architecture further breaks down into multiple phases (Table 5).

Phase	Focus
Syntactic Checks	Type checking for the component and connectors in the connections
Semantic Checks	Check for the matching of parameters (protocols) or use formal specification (second level of checks)
Linking Checks	Final checks for the matching of connections between components

Table 5: Phases of software architecture analysis

The compiling stage finally links the component by specifying linker directives and recompiles the associated component again.

The communications between components must be abstract and high level. Since WebCODS components are developed by different providers, these components are strongly decoupled. Each component in WebCODS consists of a set of Ports to interact with the outside world. The examples for high-level connections are through pipes, sockets, and files.

WebCODS is aimed to ensure that each component specified in the composed application can establish valid links together. If links between components are valid, the components are hooked up together with the minimal overhead added to the system.

In order to achieve this purpose, we have identified that an ADL is required to:

1. Identifying all basic interactions allowed by components (limited to Java)
2. Typing description of ports
3. Describing of how ports are connected to each other

The UniCon-based description of WebCODS provides information about the connectivity of the component and performs component composition. Since the connectivity of components is achieved with a set of predefined connectors, the environment can analyze the validity of connections with type-checking and the modes of connection (in, out and bi-directional).

7.2.1 Ports and Connectors

The description of the available connection mechanisms for components available in the Java environment is described in details in the following sections. Table 6 summarized the analyzed results.

Ports	Mode	Broadcasting	Supported
Procedures Calls	In, Out	No	No
Pipes	In, Out	No	Yes
Events	In, Out	Yes	Yes
Shared Variables	In, Out, In/Out	Yes	No
Sockets	In, Out, In/Out	Yes	Yes
Files	In, Out	Yes	Yes

Table 6: Summary for connections mechanism for components in Java

7.2.1.1 Procedures Calls

Components can be connected together via method invocation. This kind of connection mechanism is similar to using interfaces when programming in Java. The interface to components are predefined. The procedure definer provides actual implementation of the functions. When the procedure caller wants to use the component, the caller obtains the implementation from the procedure definer. Since the interface to the component is predefined, the caller can invoke the function with the given interface.

The connection of components via procedure results may have two scenarios:

- The definer and caller have an agreement on the interface
- The definer and caller do not have an agreement on the interface

The first case is the simple case to handle the connection. The connector uses the on-line bridge pattern to handle the connection (DeLine, 1999). The procedure caller first defines a way to pass references of the procedure definer into itself. Then the caller is able to invoke the method according to the specified interface. The bridge serves only the purpose of passing references from input component to output component.

The second case can be handled with mediator pattern (DeLine, 1999). The procedure caller and definer both specify their interfaces to the outside world. During the connection phase, the connector maps the functions required by the caller to the functions provided by the procedure definer. This step requires human intervention, but can be automated based on the matching of function signature and naming scheme. The code for the bridge must be generated at runtime to allow the function caller to call the bridge and the bridge calls the procedure definer with the

correct function (Bridge Pattern). The use of compiler and source code of all involved components is required to generating the executable of the connector.

In component-based systems, procedure calls between components creates strong coupling. Szyperski (1998) suggested that this approach is not considered as good design technique in component-based development environment. In the current version of WebCODS, the composition model will not consider procedure call as a valid connection mechanism.

7.2.1.2 Pipes (Streams)

The term Streams in Java has the same meaning as Pipes in Unix environment.

Streams in Java are separated into InputStreams and OutputStreams. However, they cannot connect to each other without a intermediate connection. The intermediate connection depends on a generic representation of data within the stream. For example, the generic form of the data can be an array filled or a file filled with byte.

Streams can be connected as

- **Buffered** – the stream contains a circular buffer of a size specified by the user. As bytes from the stream are read or skipped, the internal buffer is refilled as necessary from the contained input stream, many bytes at a time.
- **Non-buffered** – the usage of this kind of stream is similar to the buffered stream except the buffer size is 1. The contained byte will be over written with the new data received from the input stream. Therefore, data will be lost if the reading from the output stream is not as fast as writing operation.

The PipedStream available from Java presents a little different mechanism than the discussed streams. PipedStream allows an input stream directly linked with a output stream. If the input and output stream is able to connect with each other, it is not necessary to use the generic form of data representation.

In this case, first the connector must set up two piped streams:

1. PipedInputStream – input stream for piped input stream
2. PipedOutputStream – output stream for piped output stream

The stream connector uses the piped streams to connect input and out streams together. The implementation of the connector varies among non-buffered and buffered stream. The major difference between them is the kind of input streams used during the instantiation of the connector.

In the buffered version of the connector, the piped stream is first bounded with a buffered input stream in the input side for buffering. The size of the buffer is specified by the composers who create the connections between components. However, in the non-buffered version, the piped input stream is connected only with a basic input stream. Other than that, both connectors are very similar with each other.

7.2.1.3 Events

The events passing mechanism works like procedure with a well defined interface from the Java language. The Event passing mechanism in Java has defined interfaces for Events and EventListeners. The underlying mechanism for passing events to events handlers is based on the method with the following signature:

```
void addEventListener(Event e)
```

In event passing, the procedure caller is the component that generates the Event. The reference to the procedure definer is passed by the `addEventListener` method. When an event is generated, all referred procedure definers will be notified. Then, the definer reacts to the event according to their implementation.

Two connectors can be created in this category,

1. Point-to-Point connection – a one to one connection between an event generator and an event handler
2. Broadcasting – a one to many connections between an event generator and event handlers. The event generator is broadcasting the event to all the registered event handlers through the `addEventListener` interface.

7.2.1.4 Shared Variables

Since the Java Virtual Machine prohibits programs from directly accessing physical memory locations, sharing variables in memories is not possible in Java. Moreover, WebCODS deals with compiled components, so it is impossible to set up variables referred by the component which are not known at compile time. Due to the above limitations, it seems that the connection between components using shared variables is not possible.

7.2.1.5 Files

Files can be used to store output data from a component. Then the file can be read by another component as input. The connection between these components requires an agreement of the name and location of the file. Therefore, the connector of file requires the following parameters:

- Name of the file
- Location of the file

After specifying the parameters, the connector passes the specified parameters (file name) into the components. The mechanisms for output to and input from files are left for the corresponding components to handle. Since a file can be read by multiple readers, this type of Port supports both point-to-point and broadcasting.

7.2.1.6 Sockets

Data can also be passed using sockets. There are two types of sockets supported in Java language by default:

- Socket – uses Streams for input and output
- Datagram Socket – uses Datagram Packet for input and output

There is only one parameter required for Socket connection:

- Port Number (Since WebCODS is focused in local composition of components, the network address to connecting ports is defaulted back the local machine)

According to the Java documentation (Sun Microsystems, 1999), the current implementation of Socket and Datagram Socket do not supporting multicasting. The language prohibits the broadcasting capability of the connection. The mode of the connection for each type of socket can be In, Out, and In/Out.

Because of the incapability in the input and output mechanism of the Datagram Socket and Socket, they become two distinct types. The reasons are:

- The commonalties between Datagram Packet and Stream cannot be identified
- The input and output of Datagram Socket and Socket cannot be connected with each other

7.2.1.7 Supporting Connection Mechanisms

Each kind of port in Table 7 contains a description of the behaviors of different connectors supported in WebCODS (Table 8). The *Mode* of connection defines the direction of the flow of information. Event, File and Pipe support only mono-directional flows, while Sockets and Datagram Sockets support both mono- and bi-directional connections. Event, and File also support *Broadcasting*, that is the possibility of attaching multiple “in” ports to a single “out” port.

Connector	Type	Description
Event Connector	Event	Sets up the links between “Event” ports, so that an event-generating component is connected with an event-handling component.
File Connector	File	Supports the connection between two “File” ports, handling the file name and location.
Non-Buffered Stream Connector	Pipe	Establishes a pipe to connect “Pipe” ports together.
Buffered Stream Connector	Pipe	Similar to the Non-Buffered Stream Connector. In this case, the pipe has an associated buffer.
Socket Connector	Socket	Supports the connection between two “Socket” ports, handling the port number and IP address.
Datagram Socket Connector	Datagram Socket	Similar to the Socket Connector, but supports broadcasting and operates on packets.

Table 7: Connectors in WebCODS

Port	Type	Connector	Mode	Broadcasting
Event	Event	Event Connector	In, Out	Yes
File	File	File Connector	In, Out	Yes
Pipe	Pipe	Buffered Stream Connector	In, Out	No
		Non-Buffered Stream Connector		No
Socket	Socket	Socket Connector	In, Out, In/Out	No
Datagram Socket	Datagram Socket	Datagram Socket Connector	In, Out, In/Out	No

Table 8: Types of Ports supported in WebCODS

The validity of connection is enforced according to the following scheme:

1. Each port can connect only to another port of the same kind.
2. “In” ports can connect only to “Out” ports and vice versa; “Bi-directional” can be connected only to other “Bi-directional” ports.

The framework does not allow composition through implementation inheritance. The components are interconnected only with a predefined set of interfaces and connectors. These restrictions avoid inconsistencies between components and the connectivity among connectors.

The composition of the components is analyzed in a graphical environment. The composition of components is achieved by linking ports from components with the supplied connectors form the composition environment. Components and connectors can also be parameterized for adaptation to the environment during actual instantiation. The composition analysis does not involve compilers and creations of glue code for connectors and components.

7.3 Instance Level

Instance level is used to instantiate components which exist in the composition level. Components in need of instantiation require both specification and executables. The instantiation is basically interpreting the specification and constructing instances of composed components.

Instantiation of components in WebCODS may refer to constructing running instances of both primitive or composite components. The steps for instantiation vary between primitive and composite components. The process varies because primitive components relate directly to executables, while composite components refer to a collection of components which can be primitive or composite.

The instantiation of primitive components is the direct execution of the corresponding binaries. The instance of the component can be parameterized with the parameter supplied in the composition level. When the instantiation occurs, these specified parameters are passed into the component as runtime variables. The component uses these parameters to construct a parameterized instance of the component from the executables (Figure 5).

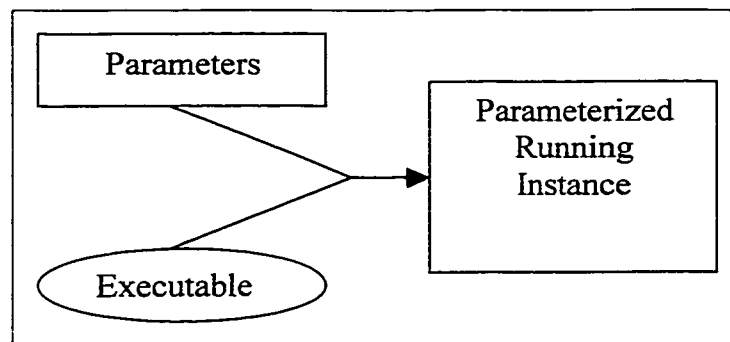


Figure 5: Instantiation of Primitive Component

Instantiation of composite components is a recursive process since all the referred components need to be instantiated. The first step of the instantiation transverses the composite components to execute of all referred primitive components. After actual instances of programming features are running, the required connectors used to connect ports among the executed components are instantiated with the specified parameters. The connectors are ready to be used to connect components together according to the specification. When the instantiation of the current level of components is finished, the process is ready to move on to the above level that encapsulates the current level of components. The process repeats until it finishes instantiation of highest level of composite components.

Figure 6 shows the steps required to instantiate the composite component CCA (Composite Component A). The component CCA encapsulates another composite component CCB (Composite Component B) that refers to a primitive component PCC (Primitive Component A). The first step of the instantiation is to transverse CCA to identify and execute all primitive components. After PCC is instantiated, the next step is to instantiate CCB. The execution of CCB establishes a connection between CCB and PCC. Then the process moves to the next level and establish another connection between CCA and CCB. The instantiation of CCA ends the recursive process because the highest level of the composite components is reached.

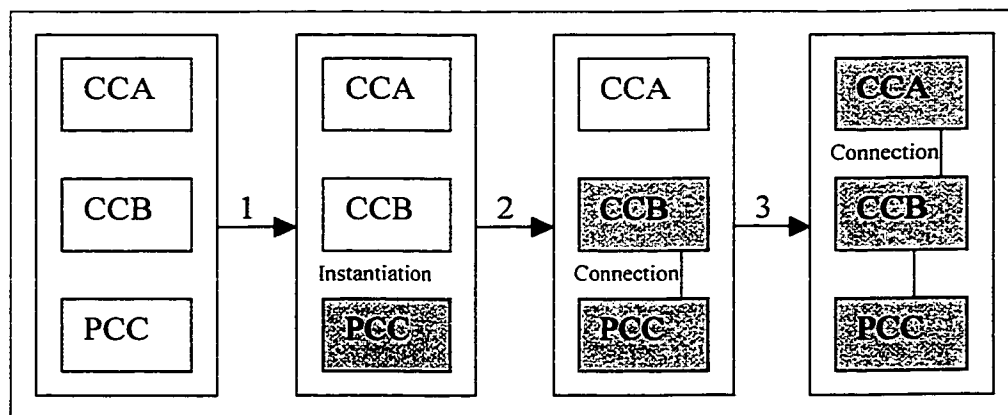


Figure 6: Instantiation of Composite Components

A flexible application can be achieved, if its architecture allows its components to be removed, replaced and reconfigured without perturbing other parts of the application. This kind of application can be generated, if the instantiation of the application does not require recompilation of components and connections. There are no new binaries required to load into the system for executions and reestablish connections among components. In this way, the software architectures can be altered at runtime by modifying the connectors in the graphical component composition environment. To support dynamic composition of components, the environment should allow components to be loaded and unloaded into the framework dynamically.

The reuse of the same application structure is supported by saving the structure of the composed application. The save involves the architecture description of the application in the UniCon-based ADL description. The description is created by analyzing the structure of the composed applications in the environment. The description specifies the required components for execution of this component and all the connection mechanism among those required components. The description is in text format and it does not contain any runtime-time information about the application.

The saved description can be reloaded into the environment as a composite component. The loading of saved components first sends the description to the composition level. Then the composition level evaluates the composition mechanism specified in the description, and validates the rules. If the validation of the description succeeds, the component will be forwarded to the instantiation level for execution.

Chapter 8: Component Brokerage

The broker in WebCODS supports component brokerage by allowing users to identify suitable components and downloading them. The components in the database are represented with facets and free-text description. The use of facets allows classifiers more freedom to create complex relationships by combining facets and terms. It is also much easier to modify than other classification schemes, because one facet can be changed without affecting others in the classification scheme. The free-text description is provided as an auxiliary classification scheme. Classifiers can use the free-text description to supply more detail for their components. The identification of components in the database is primarily based on facets and uses keywords in the free-text descriptions as the secondary searching media.

The classification of components in WebCODS is the responsibility of providers. When providers want to announce new components to the broker, they classify the new component using facets and free-text descriptions. The components being sent to the broker contains:

1. Binaries for execution
2. UniCon based description for connectivity and internal structure
3. Classification details for the component

When the broker receives components from providers, the classification details of components are extracted and stored in the search engine in the broker. The client uses the search interface provided in the client environment to query the component database using the search engine. The search engine returns a set of matched components' classification details to the client. The architecture of the component brokerage environment in WebCODS is shown in Figure 7.

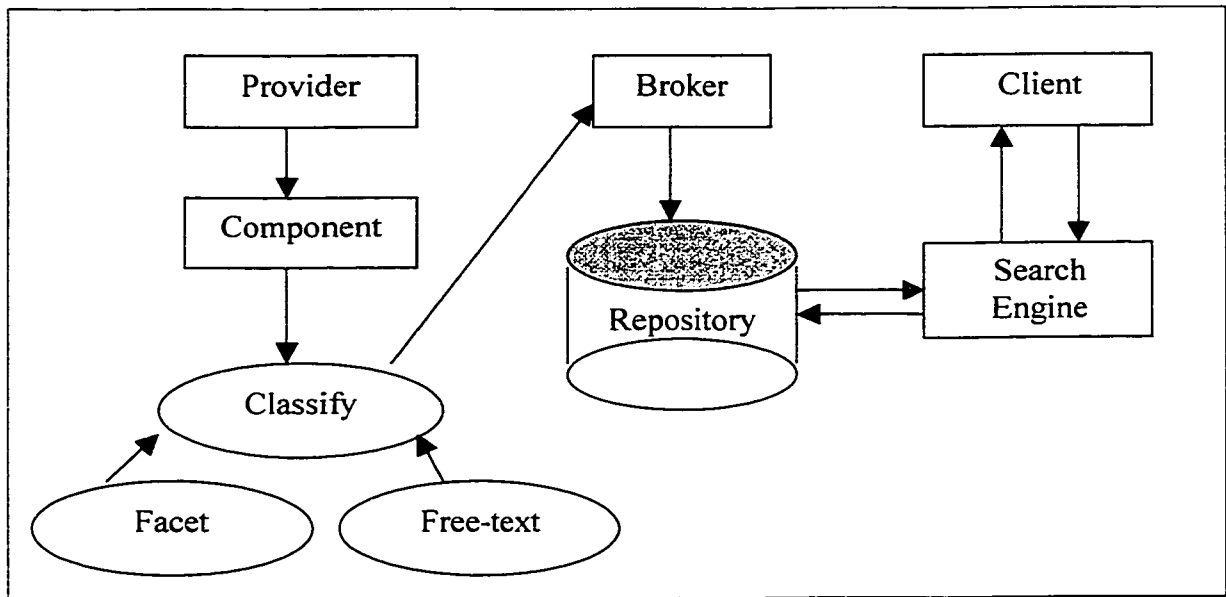


Figure 7: Component Brokerage Environment in WebCODS

8.1 Component Classification

The classification method selected for WebCODS component exhibits common requirements of component libraries (Li *et al.*, 1998).

1. The classification information must express relevant information about the component from the users point of view.
2. The classification strategy must be applicable to a broad spectrum of components.
3. The classification must not be too complex. High complexity method will decrease the quality of the classification scheme since more work is required in order to understand and use the classification method properly.

The faceted classification method in WebCODS defines a set of independent facets that describe essential features of components. The facets are applicable for all components in the component library. Each facet is associated with a structured set of terms. The facets form a list of pairs: facet, and the term value assigned for the facet. Each term may be associated with several synonyms specified in a thesaurus.

The base facets are orthogonal and independent of each other. They reflect sufficiently the characteristics related with the use of components.

The base facets are listed in descending order of their importance as follows:

- **Domain:** The names of domain that the component is used or may be used. The term “domain” characterizes a set of similar systems or applications that share some common functionality.
- **Action:** The function that is performed by the component.
- **Platform:** The software platform that is used to execute the component.
- **Safety:** The safety level of executing the asset.
- **Needed Resource:** Other related sources required by the successful execution of the component.
- **Cost:** The cost of using the component charged by the broker.

If the provider wants to add in new facets, the new facets are added to the bottom of the base facets list in the order specified by providers. The new facet list is presented to the broker which then updates the facet list stored in the search engine. When users query the broker for components, the search engine presents the updated facet list for users to query the available components in the broker.

The determination of term space of facet is difficult because a legalized term standard is absent. The tentatively term space is presented below, which needs to be modified and extended while the classification method is being used. The provider environment used for classification allows classifier to enumerate the available term space for each facet. Providers of component may provide new terms at any time when they provide new components. The current supplied term spaces for each facet are listed in Table 9.

Facet (Description)	Term Space (Value)
Domain	"Metric Collectors", "Data Analyzer", "Data Viewer"
Action	"Analyze", "Extract Data", "View File", "Convert File"
Platform	Java 1.1, Java 1.2
Safety	"High", "Medium", "Low"
Needed Resources	"Analyze", "Extract Data", "View File", "Convert File"
Cost	10, 20, 30

Table 9: Samples of Facets and Term Spaces used in WebCODS

The administrator of the broker maintains the terms of facets in the classification scheme. In order to maintain consistency of term space, if terms supplied by providers of components are not coincident with existing terms in the scheme, the administrator will be consulted to add new terms and their synonyms into the thesaurus.

The component classification uses faceted method as the main classification strategy, and uses free-text description as auxiliary method. In the description, the provider explains the behavior of the component in natural language. When users retrieve the component, they obtain more understandable details from the provided text description than with facet classification alone. Since the description also contains controlled and uncontrolled keywords, it can be used for matching queries supplied by users.

8.2 Operations of the Broker

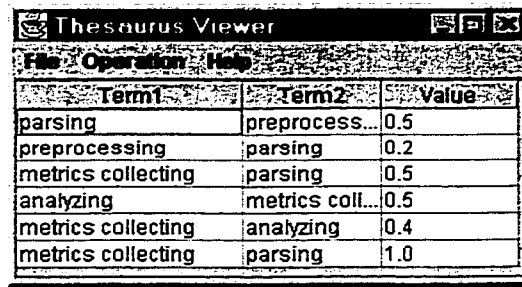
The broker maintains a storage area for the submitted components from providers. Components submitted by providers contain the core elements for component composition and the corresponding classification information. The classification information contains a facet list and free-text description.

When a client queries the component broker, the search engine provides two modes of searching. The broker can provide the client a list of currently available components. The returned list contains only the classification details for components. If clients are interested in downloading the component, they have to make a separate request for downloading. The broker also allows the client to search the component database with a query. The search generates a ranked result set which match the query.

The search engine is accompanied with a thesaurus. The search engine uses the thesaurus to extend the exact-match paradigm in searching. The thesaurus contains a list of synonyms used in the system-defined facets. For each synonym contained in the thesaurus, there is a ranking of how close it is related to each other. In this way, the search engine is able to identify one or more related components with ranked closeness according to the input query.

Figure 8 shows a sample thesaurus for the vocabulary used in the Action facet. The weighted relations between synonyms are also displayed. For example, the term “metrics collecting” and “parsing” are synonyms in the domain; the assigned weight for their closeness value is 1.0. The relationship between terms is not bi-directional. Considering the synonyms “parsing” and “preprocessing”, if users refers to the term “parsing”, preprocessing may be considered as an activity that involve parsing. However, users interested in the preprocessing actions may not have a high interest in

parsing. Therefore, the closeness from the term “parsing” to “preprocessing” is stronger than the opposite relationship.



Term1	Term2	Value
parsing	preprocess...	0.5
preprocessing	parsing	0.2
metrics collecting	parsing	0.5
analyzing	metrics coll...	0.5
metrics collecting	analyzing	0.4
metrics collecting	parsing	1.0

Figure 8: Thesaurus for Action Facet

When a query is submitted to the broker, the search engine expands the criteria using the thesaurus. The expanded query contains all the related synonyms according to the relationship in a specified query. The expanded query is also ranked according to the closeness to the original query. Searching of components is based on the exact match of terms in the expanded query. Since the query set is ordered, the result set is also ordered according to the expanded query.

The engine takes in a query that contains two parts: one for facets and one for free text description. For each facet specified in the query, the engine checks the thesaurus and expands the submitted facet value with the list of weighted synonyms. The engine selects the components that match with the expanded query. Components that match the query are added to the result set with their rank, according with the weight in the list of synonyms. If a search is required in the free-text description, the component descriptions not containing the query term are removed from the result set constructed in the previous stage. The set returned by the search is all the possible matched components with ranking.

8.3 Security

WebCODS presents different security policies: password authentication, signed message and encryption. The techniques that we will use are based on public, private and Diffie-Hellman Key. The communication protocol can be subdivided into 3 stages as shown in Figure 9: registration, initialization and transferring data.

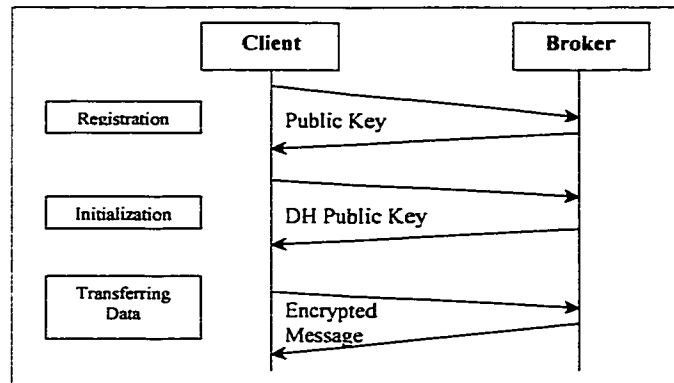


Figure 9: Security protocol used in WebCODS

The registration stage allows clients to register with a system administrator in WebCODS. The client first generates a set of public and private key. The created public key will be sent to the administrator. The administrator then creates a certificate for the received public key. If the registration is success, the administrator will deliver the created certificate and the system certificate to client. After the installation of certificates in the key management system, the registration step is no longer required until the certificate expires.

The initialization step begins after the client connects with the broker as an authenticated user. The client application generates a pair of keys with Diffie-Hellman Key agreement. The created public key is encrypted with the broker's

public key and signed with the private key of the client. After receiving the encrypted key, the broker then verifies the signature of the sender and obtains the key in the message. Then, the broker follows a similar approach to send its Diffie-Hellman public key to the client. At this point, both clients and the broker have obtained each other's key for secure data transfer.

The transferring of data in WebCODS is based on the JavaSpaces architecture. Dispatching components on a public network have to address two different problems: send the right component to the right receiver and protect the content of the message to be read only by the authorized parties. The entry used to deliver messages in the space of WebCODS features a destination and data section. The destination section contains the delivery information of the message. The data section stores the encrypted contents of the message. The message is encrypted with Diffie-Hellman Keys, therefore, only authorized parties are able to decrypt the message.

Chapter 9: Implementation

Since WebCODS targets the Web-based environment, it requires its components to be executable in different machine platforms. As suggested by (Faison, 1997), Java satisfies this requirement.

The implementation of the WebCODS system includes (Figure 10):

- a. Composition environment to compose components to form applications,
- b. Search engine for locating matching components for composition and
- c. Security center to provide protection for transferring components among the broker, clients and providers.

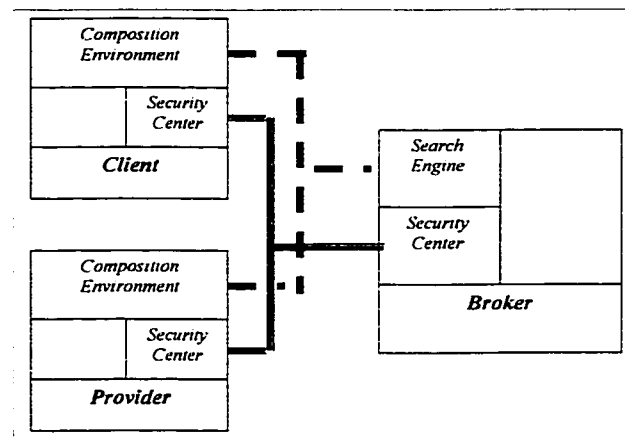


Figure 10: Distribution of components in WebCODS

The detailed descriptions of implementation strategies to support the above features are described in the following sections.

9.1 Composition Tool

The components to support component composition are: (1) an interpreter that processes and analyzes connectivity description for components, (2) a tool used to transfer components in the network, and (3) a graphical composition environment for manipulating components.

9.1.1 Interpreter

Components in WebCODS are accompanied with a connectivity description. The descriptions are analyzed by a textual interpreter generated with JavaCC. The target product of the interpretation is a set of object-based representation of the structure of the component. The OO-based representations are used in the instance level for instantiation and future composition.

Components in WebCODS are accompanied with a description. The description characterizes the structure of the component into components, connectors and ports. To convey this information, detailed descriptions for the internal structure and the connectivity of the components are required. The internal structure reveals how components encapsulated in the components are connected together. The connectivity describes the incoming and outgoing interfaces available inside the component. UniCon has been adapted to support runtime composition of components. The modified version of UniCon contains only two major building blocks: (1) a Port Specification, and (2) an Implementation Section.

The syntax of the modified UniCon language follows in BNF format is shown below:

```
Component ::= COMPONENT ComponentName  
                (PortList)*  
                [Implementation]  
                END COMPONENT
```

```
PortList ::= PORT PortName MODE aMode IS Type  
            MEMBER(StringWithMemberName)  
            END PORT
```

```
Implementation ::= IMPLEMENTATION  
                    (INSTANCE      InstanceName      =  
                    ComponentName [Parameter] )*  
                    (Connect)*  
                    END IMPLEMENTATION
```

```
Connect ::= CONNECT PortName TO PortName (, PortName)*  
            TYPE Type IS ConnectionMethod  
            [PARAM attribute] [Parameter]
```

```
Mode ::= IN | OUT | INOUT
```

```
Type ::= EVENT | FILE | PIPE | SOCKET | DATAGRAMSOCKET
```

```

ConnectionMethod ::= EVENTCONNECTOR | FILECONNECTOR
                    |
                    BUFFEREDCONNECTOR
NONBUFFEREDCONNECTOR
                    |
                    SOCKETCONNECTOR
                    |
                    DATAGRAMSOCKETCONNECTOR

```

```

Parameter ::= { Parameter (, Parameter)* }

```

ComponentName, *InstanceName*, *PortName* are the usual C-like identifiers.

StringWithMemberName is a string with the name of the method to use in it.

Parameter can be a string or “\$” <DIGIT> for referring to the index of the instantiation parameter.

Figure 11 shows the UniCon-based description of the system illustrated in Figure 4 (A, B, and the composed system AandB), assuming that all ports are of type Pipe.

<pre> COMPONENT A Port ζ MODE OUT IS Pipe END PORT Port ω MODE IN IS Pipe END PORT END COMPONENT </pre>	<pre> COMPONENT B Port ψ MODE In IS Pipe END PORT Port ξ MODE In IS Pipe END PORT END COMPONENT </pre>
<pre> COMPONENT AandB Port C1.ω MODE OUT IS Pipe END PORT Port C2.ψ MODE OUT IS Pipe END PORT IMPLEMENTATION INSTANCE C1 = A INSTANCE C2 = B CONNECT C1.ζ TO C2.ξ TYPE Pipe IS NonBuffered END IMPLEMENTATION END COMPONENT </pre>	

Figure 11: Connectivity description for the components shown in Figure 4

Components referred by INSTANCE and CONNECT keywords can be either primitive or composite. The composite description keeps only references to subcomponents and allows subcomponents to specify their implementation details. The description of a component can be interpreted into a set of Java objects. The representation can be used for both types of components. Descriptions are stored in a wrapper object called “Typed Component”. This wrapper object is used to store (1) the descriptions of all referred subcomponents (2) the information of the interfaces, (3) the connectivity information within the abstract component. The hierarchy of the Typed Component is represented in UML in Figure 12.

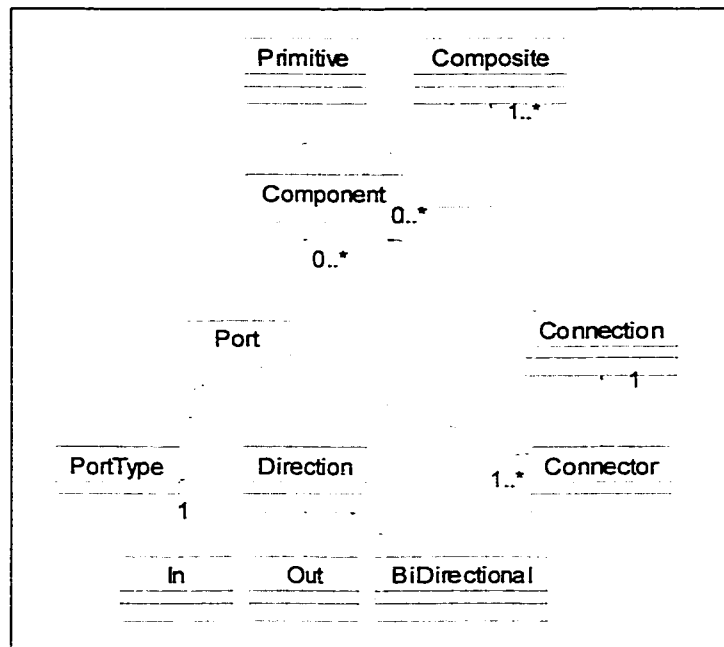


Figure 12: UML diagram shows the structure of components in WebCODS

The description of a component is transferable in the network with the object serialization provided with Java. The transfer of component description also transfers the description of all subcomponents.

9.1.2 Transfer of Components

The transfer of components requires the moving of components' executables and the corresponding description.

When constructing the OO-based representation of components, the component referred by the INSTANCE can refer to an actual instance of the component. The interpretation of the description can establish connections directly between instances of components. Using the serialization properties of Java, the states of components and their connection properties can be saved and reconstructed via deserialization of the OO-based description. This approach is currently being used in JavaBeans.

However, implementation of transferability of components in WebCODS based-on serialization raises feasibility issues:

- Components can be serialized only if all referred classes implement the Serializable interface.
- Some connectors require reconnections in the target environment, such as, Files and Socket
- For deserialization to be successful, all implementing classes must be concurrent and available on the target machine.

For these reasons, we cannot restrict WebCODS component to be based on the serialization properties of Java. The solution is to use the notion of an “Unresolved Component”. The Typed Component contains references to WebCODS components in the INSTANCE construct in the description. Instead of referring to the actual instance to the components, we replace the reference with a symbolic reference -- “Unresolved Component”; a reference that will be resolved on the client machine. Unresolved components are then serialized and sent to client machines.

On the target environment, the OO-based description with unresolved components is deserialized. This forms the “Reference Component.” The Reference Component is used with both primitive and composite components. When the Reference Component refers to a primitive component, the concrete implementation stored in the object refers to the start up Java class. When the typed component represents a composite component, the component stores references to a collection of other Reference Components. Establishment of connections can be done among instances of components. The Reference Component contains the runtime information about (1) the implementation of the component and (2) the types of ports.

9.1.3 Composition Environment

Composition environment supports analysis of connectivity, and composition and instantiation of components. The composition environment provides a set of predefined connectors for connections among components. The composition and instantiation of components enables the user to add in new components to the environment and execute them.

The interfaces of components are defined with ports. Each port has a name and a type. Available types are Event, File, Pipe, Socket and Datagram Socket. The connections are typed; each kind of ports has a specific connector to link them together. Currently, WebCODS does not handle other kinds of interconnection mechanisms proposed in the literature (Garlan, 1997; Deline 1999) such as procedure calls and shared variables.

Using a predefined set of connectors allow connectors in WebCODS to be written with the reflection property in Java and the template-method design pattern (Gamma et al, 1994). Each type of connector has a template to handle the connection. Recompile of components and connectors is not required when new connections are established components. Runtime modification of software architecture is supported.

Consider the previous example in Figure 4. Port ζ of A and port ξ of B are connected with a non-buffered stream connector. Component A uses the method `setOutputStream` and B uses `setInputStream` to set up the reference to the pipe established by the connector. The interfaces of components are not predefined as in a framework.

Figure 13 details a piece of Java code that could implement the required connection. However, this approach requires the generation of glue code for the connector and linking of components at runtime with the name of the actual procedures to be called, `setOutputStream` and `setInputStream`.

```
public class MyConnector extends NonBufferedConnector {
    ComponentA object1;
    ComponentB object2;

    InputStream is;
    OutputStream os;

    public MyConnector(ComponentA object1, ComponentB object2) {...}
    ...
    public void connect() {
        //Set up reference to the Pipe initialized in the connector
        object1.setOutputStream(os);
        object2.setInputStream(is);
    }
}
```

Figure 13: Possible Java code for the connection of A and B in Figure 4.

Figure 14 illustrates the proposed solution for the Non-Buffered Connector. The component specific codes in Figure 13 have been replaced with source in the reflection package of Java to support generic components. Parameters for connectors are not constrained to predefined values, they can be specified in the visual composition environment or loaded from the specification language.

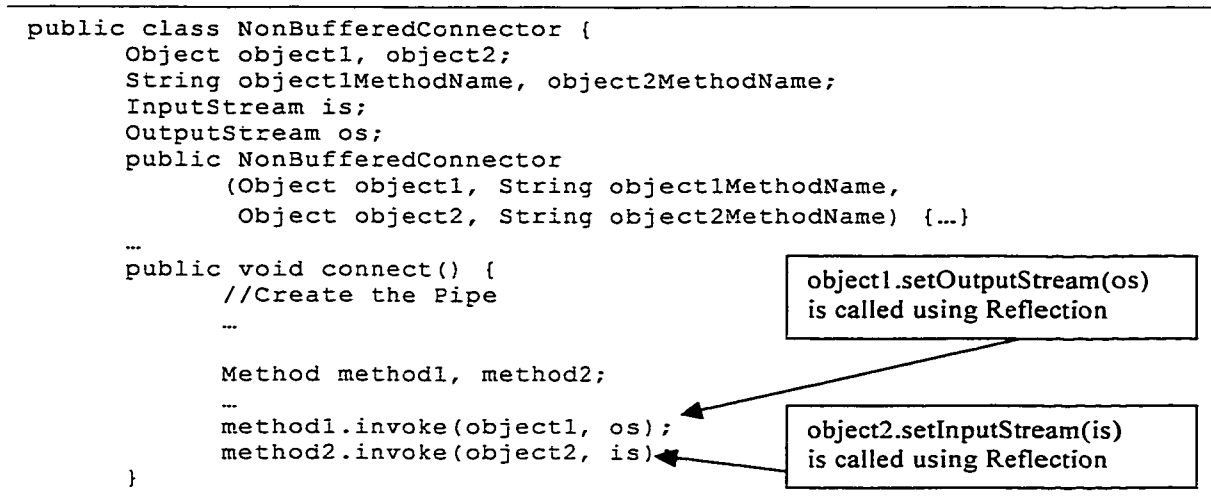


Figure 14: Template used by the Non Buffered Connector

The composition environment allows dynamic loading of component descriptions into the environment. The connectivity information associated with the component is converted into OO-based descriptions. The analysis of connectivity is based on reflection to validate the correctness of connections with typing-checking.

The instantiation of components converts unresolved components into resolved components. The resolved component contains references to the actual instance of the corresponding component. The instance of the component is finally executed and awaiting for further composition in the environment.

Usually binary components can be instantiated with different parameters by calling constructors with different interfaces. The instantiation of component with parameters requires the framework to provide a predefined function of factory design pattern to create instances of components. The method takes in an array of Strings as parameters and it has the following interface:

```
Object createObjectWithParameters (String[] args)
```

Both primitive and composite components can be specified with parameters. In the case of primitive, the specified parameters are passed to the actual instance of component. Specifying parameters for composite component requires an extra step to distribute the supplied parameters to each referred component. The description should contain directives for distribution of these parameters. At the end of each instance and connect statement of the textual description, there is an instantiation section. The section is used for specifying the parameters for instantiation. The parameter can be final or referencing to a runtime parameter specified during instantiation. We have adopted an indexing scheme for referencing of parameters. The parameter “\$1” uses “\$” to signal the use of the indexing scheme and “1” to refer to the first parameter supplied at runtime.

For example, two primitive components (File Browser and File Content Viewer) are composed to form a Complete File Viewer. Each primitive component is displayed in separate windows. The display windows are differentiated by their titles, which can be supplied as input parameter to the components. Therefore, the composed component requires two titles. The first parameter is the title for the File Browser component (\$1), and the second parameter is the title for the File Content Viewer component (\$2).

When the Complete File Viewer component is executed with the parameters “Browser” and “Viewer”. The instantiation of the composite component binds the parameter \$1 with “Browser” and the parameter \$2 with “Viewer”. Each parameter is passed on to the corresponding component for instantiation. The bounded parameters are passed on to the sub-components. The next instantiation step rebinds the argument “Browser” as \$1 in the File Browser component, and the parameter “Viewer” as \$1 in the File Content Viewer component. The execution of the primitive component uses the supplied parameter for instantiation.

The textual description of the current application architecture can be obtained by the analysis of the composition environment. The generated description can be saved and loaded into the environment for later retrieval.

9.2 Search Engine

The classification details of components to be used by the search engine are stored in the eXtensible Markup Language (XML) format. XML is a markup language much like HTML. It is designed to describe data. It uses a DTD (Document Type Definition) to formally describe the data. XML can also be used to store data in files or in databases. Applications can be written to store and retrieve information from the store, and generic applications can be used to display the data.

The component classification information in XML is supplied by component providers. The supplied description is verified by a predefined a DTD by the broker. The verified XML descriptions are stored in the broker and they will be used by search engine. When the XML description is provided to clients, they can display the document in a tree view to visualize the enclosed information.

The purpose of a DTD is to define the legal building blocks of an XML document. It defines the document structure with a list of legal elements. The DTD used in WebCODS is shown in Figure 15. The DTD specifies the XML description will contain the following sections:

1. Name – the name of the component.
2. Description – the free text description of the component in natural language.

3. Facets – the set of facets used to classify the component and their associated attributes. The facets are listed in descending order according to their importance.
4. Ports – list of the unconnected ports of the component.
5. Provider Name – the name of the component provider.

```

<!ELEMENT List (Components*)>
<!--ELEMENT Components (#PCDATA|Facets|InputOutput|Provider) *-->
<!--ATTLIST Components Name CDATA #REQUIRED
Rank CDATA ""-->
<!--ELEMENT Facets (#PCDATA)-->
<!--ATTLIST Facets Name CDATA #REQUIRED-->
<!--ELEMENT InputOutput (Port*)-->
<!--ELEMENT Port EMPTY-->
<!--ATTLIST Port PortName CDATA #REQUIRED
PortType CDATA #REQUIRED
PortDirection (IN|OUT|INOUT) #IMPLIED-->
<!--ELEMENT Provider (#PCDATA)-->
<!--ATTLIST Provider Name CDATA #REQUIRED-->

```

Figure 15: The DTD for the XML description of WebCODS component

A sample XML description of a WebCODS component is shown in Figure 16.

```

<?xml version="1.0" encoding="UTF-8"?>
<List>
  <Component Name="TableMetricsViewer">
    <![CDATA[Table-based visualization of the generated metrics data]]>
    <Facets Name="Domain">MetricsExtraction</Facets>
    <Facets Name="Action">Viewing</Facets>
    <Facets Name="Platform">1.1.7</Facets>
    <Facets Name="Safety">low</Facets>
    <Facets Name="NeededResource">dataInput</Facets>
    <Facets Name="Cost">10</Facets>
    <InputOutput>
      <Port PortName="FileInput" PortType="File" PortDirection="IN" />
    </InputOutput>
    <Provider Name="Component Provider A"></Provider>
  </Component>
</List>

```

Figure 16: A sample XML description for WebCODS components

With all the classification details for components encoded in XML, the XML description will be used by search engine in retrieving of components. The implementation of the search engine contains two parts: the thesaurus and the engine.

The thesaurus contains terms and weighted relation between allied words. The contained information in the thesaurus can be represented as a graph. Each node in the graph represents a term and each edge represents the directed weighted proximity relationship. The weighted graph is used to help identify related terms. When a query is made, the graph is consulted to find all related terms and returns a related set of terms used for querying. Weights are assigned to the edges, the higher the weight

assigned to the edge connecting the two terms, the closer the perceived proximity of the terms.

Using a node as entry point, the traversal of the graph results with a list of synonyms related to the term represented by the entry point. For each synonym in the list, there is also a weight that explains how the two words are related.

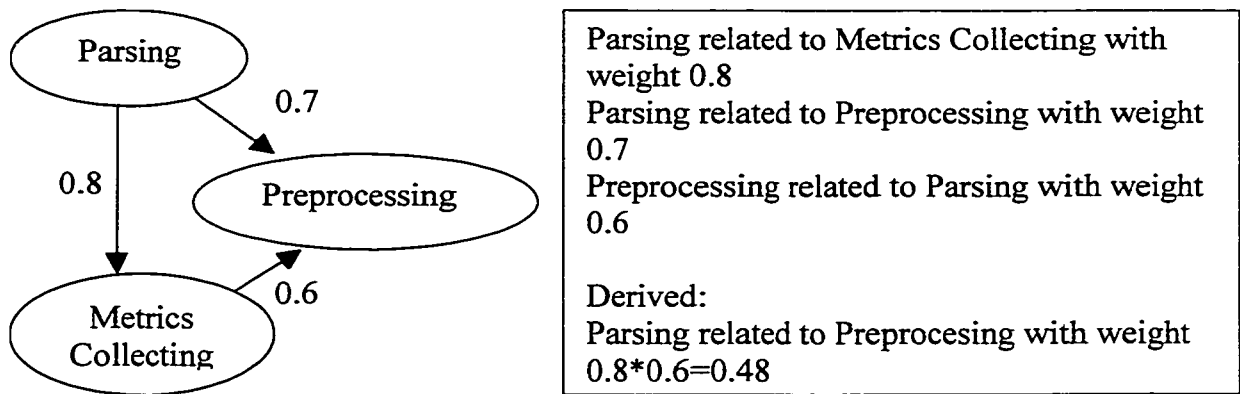


Figure 17: Example of relations in the Thesaurus

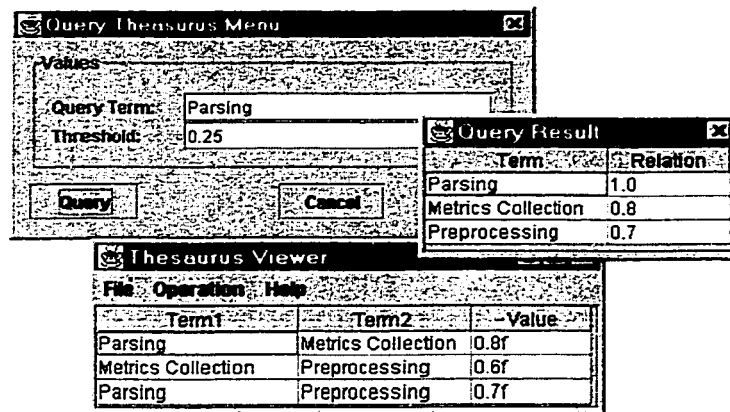
Figure 17 shows some sample relations that might exist in the thesaurus. Parsing, Preprocessing, and Metrics Collecting are terms in the thesaurus. When consulting the graph for related synonyms for Parsing, the graph will be transversed starting from Parsing. The querying of the graph finds that Metric Collecting relates to Parsing with weight 0.8 and Preprocessing relates to Metrics Collecting with weight 0.6. The proximity relationship between Preprocessing and Parsing is the products of weight from Parsing and Metrics Collecting, and Metrics Collecting and Preprocessing. The function used to calculate the proximity relation is shown as:

$$w(\text{Parsing}, \text{Preprocessing}) = w(\text{Parsing}, \text{Metrics Collecting}) * w(\text{Metrics Collecting}, \text{Preprocessing}).$$

When there is more than one value in the relationship from two terms, so the system has to choose the maximum weight relations:

$$w(term1, termN) = \max_{all\ the\ possible\ path} (w(term1, termN))$$

In this way, all the terms may be related to the query term with different proximity relations. If two terms have a proximity relation below a threshold value (e.g., 0.3), the two terms are not related to each other. According to the above formula, the weight from Parsing to Preprocessing is maximum of the set {0.48, 0.7}. Therefore, the proximity relation between Parsing and Preprocessing is 0.7. Table 10 shows the returned set of using the term Parsing to query the graph.



Term	Relation
Parsing	1.0
Metrics Collection	0.8
Preprocessing	0.7

Term1	Term2	Value
Parsing	Metrics Collection	0.8f
Metrics Collection	Preprocessing	0.6f
Parsing	Preprocessing	0.7f

Table 10: The result set after querying the graph with the term Parsing

The engine supports query of terms in facets and free text.

When searching for terms in the facets section, the engine first expands the submitted facet value with the thesaurus. The querying of the thesaurus returns a list of weighted synonyms to the engine. The engine then identifies the components that matched with the expanded query. Since the list of synonyms is ranked, the matched components are also ranked according to the weight associated to the synonyms. The

primitive matching criteria in the engine are based on exact matching. However, for some facets that support ordinal scale, such as “Platform”, “Safety”, and “Cost”, comparison operations like equality, greater than, and lower than are also supported. The searching of facets is ordered in the same way as the importance of facets.

Searching the free text section serves as the auxiliary mechanism in WebCODS. When the component description is received in the search engine, the free text description of components is converted into a significant list (Frakes and Poles, 1991). This significant list is constructed from the free text description by discarding the useless words. The discarded words are articles, prepositions, and some other common words. To obtaining a higher accuracy matching, the list of significant words can be improved with statistical analysis to discard the unnecessary terms. The searching of free-text can also be accompanied with a thesaurus to expand the querying for free text descriptions into a list of weighted synonyms. The lists of significant words for components are compared against the expanded query for exact match. Components that match the query are added to the result set with their rank, according with the weight to the list of synonyms.

9.3 Security Center

To login to the broker, users have to supply their username, their company name, and their password. This information is used to build the MD5 hash code used for authentication. The password is not transmitted over the network, because it is used only for building this code. The username, company name and the generated hashcode for password of the user is passed to the broker for validating the user’s identity.

The validation process in the broker then uses the usernames, and company names supplied by the customer to identify the password of the customer in the database.

The obtained information is used to generate a MD5 hash code for the user. The verification of the identity is based on the matching of the two codes.

Clients and providers use the broker as an intermediary in their communications. The broker holds a public key for each customer, and each of the customers also holds the public key of the broker. Therefore, messages send among parties can be verified if they are signed by their senders.

The encryption technique used in WebCODS is based on “Diffie-Hellman” key agreement protocol. The protocol overcomes the necessity of using symmetric keys for encrypting and decrypting messages. However, the DH keys are currently not supported by X.500 certificate standard. Therefore, public DH keys cannot be transferred in the format of certificate. Since WebCODS uses DH keys for encryption, an addition step is required for the clients and broker to exchange their public DH keys.

After the user is recognized as an authenticated user, the program creates a new set of keys using the “Diffie-Hellman” key agreement protocol. After the keypair is generated, the broker and clients will exchange their DH public key. The message used to enclose the key is signed with the private key of the sender. After verifying the originality of the received public key from the broker, the client is able to build a secret key. The secret key is built with the private key of the client and the public key of the broker. The created secret key is served as a seed to build a “TripleDES” key for encryption. The algorithm uses multiple DES keys to perform three rounds of DES encryption or decryption; the added complexity greatly increases the amount of time required to break the encryption. The encrypted message can only be decrypted with the secret key generated from the public key of the client and the private key of the broker. Figure 18 shows the events occurred after the clients initialize a login request to the broker.

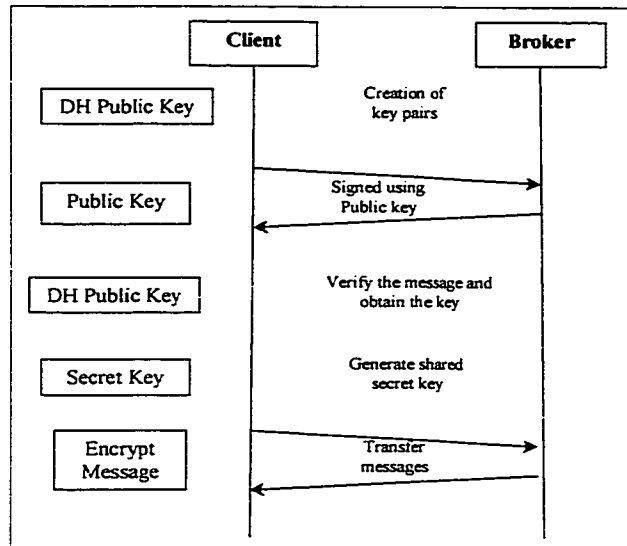


Figure 18: Transferring of keys between clients and the broker

For performance consideration, the generated keypair will be saved in the local environment for future use. The code used for decryption and encryption is shown in Figure 19.

```

public byte[] decrypting(byte[] buf, Key secretKey)
throws Exception{
    //Create a TripleDES engine
    Cipher c= Cipher.getInstance("DESede");

    //Setup up the decryption with the key
    c.init(Cipher.DECRYPT_MODE, secretKey);

    //Perform decryption
    return c.doFinal(buf);
}

public byte[] encrypting(byte[] buf, Key secretKey)
throws Exception {
    //Create a TripleDES engine
    Cipher c= Cipher.getInstance("DESede");

    //Setup up the encryption with the key
    c.init(Cipher.ENCRYPT_MODE, secretKey);

    //Perform encryption
    return c.doFinal(buf);
}

```

Figure 19: Code for decryption and encryption in WebCODS

When transferring messages to the open JavaSpaces architecture, the system adopts the following policies to ensure security. Each message posted to the space is specified with recipients; so that messages will not be delivered to unauthorized readers. Figure 20 shows the protocol for publishing messages in JavaSpaces for WebCODS. In addition, the messages are signed with senders' signatures using their private keys. Therefore, the sender of the message can be verified. The content of the message is encrypted using the sender's DH private key and receiver's public key.

```
public class WebCODSEntry implements
net.jini.core.entry.Entry {

    //Attribute used for matching
    //the receiver of the message
    public Receiver receiver;

    //Content of the message
    public Object = signed_encrypted_object;
}
```

Figure 20: Protocols for transferring messages in WebCODS

Therefore, every message dispatched on the space is encrypted and signed. Parties other than the specified recipients are not able to view the content of the message. The message senders can be verified for validity and audited by the receiver for the content of the message.

9.4 Fault Tolerance

WebCODS has a single point of failure: the broker. The broker program is responsible for all the component brokerage related activities. If the program fails, the entire WebCODS application cannot function anymore and all components submitted from component providers may be lost. The possible error recovery actions can be:

1. Backup the contents of the broker – reduce the lost of data cause by the failure
2. Provide duplicate brokers – increase the reliability of the system

The current implementation of WebCODS broker implements a backup mechanism for restoring the state of the broker before a failure occurred.

The assets in the broker are components. These components are composed of (1) XML descriptions, (2) connectivity descriptions and (3) executables. The component center in the broker first stores all these details for each component received from providers. Then the XML description will be analyzed and the analyzed result is used to form the search engine.

In the broker, the backup mechanism activates every 10 minutes to save the contained list of assets. The saving stores the three necessary elements to form WebCODS components. If the broker failed, the saved list of components can be loaded back rebuild the component library in the broker.

The rebuilding of the broker simulates the process of submitting components from providers. When a new component is added to the broker, the XML description is analyzed and the classification description is passed on to the search engine. The component library in the broker stores the connectivity description and the executables of the component.

With the given XML descriptions, connectivity descriptions and executables of saved components. Each component can be added to the reinitialized broker sequentially. In this way, the component list of the broker and the component library is restored. By adding components to the broker, the status of the search engine is also restored to the state before the broker fails.

The code used for the restoration of the broker is shown in Figure 21.

```
public void restoreBroker() {  
    //Create a new search engine  
    SearchEngine= Engine.getEngine();  
  
    //Load the saved file  
    ObjectInputStream ois = loadSavedFile();  
  
    //Obtain the save list of component  
    Vector componentList = (Vector) ois.readObject();  
  
    //Build the component list in the broker  
    broker.addComponentList(componentList);  
  
    //Build the searchEngine  
    searchEngine.addComponentList(componentList);  
}
```

Figure 21: Code for restoration of the broker

The JavaSpaces is only used as the transfer medium, so it is not necessary to be backed up. The restart of the broker does not have any interaction to the transferring medium.

Chapter 10: Example Use of WebCODS

Participants in WebCODS are separated into three parties: broker, provider and client. This section will use an example to illustrate how WebCODS involves all participants.

10.1 Providers

The component broker in WebCODS is a server program. The server provides an administrated entry point for providers and clients entering the virtual market.

After connecting to the broker, providers can use the application to create new primitive and composite components. The environment contains a UI for providers to characterize components and generate their descriptions automatically. The upload of components to the broker can be done after all required parameters are specified. Providers use the environment to create new primitive and composite components. The environment contains a UI for providers to characterize components and generates the description automatically. After the classification parameters are specified, the components can be uploaded to the broker.

For example, a provider specializing in software metrics extraction has two primitive components to offer the broker: (a) a C/C++ Preprocessor and (b) a C Software Metrics Extractor. The details of these components are shown below. The bytecode of these components are bundled in Jar files saved in the provider's local environment.

The preprocessor is the front end of a C/C++ based language compiler. It processes compiler directives and outputs preprocessed code in a stream. The main executable class for the Preprocessor component is the `Preprocessor` class. The preprocessed code is output to an output stream through an “out” pipe port `-PreprocOutput`. The reference to the output stream is set by the method `setOutputStream`. The component accepts a parameter to display as the name of the frame.

The Software Metrics Extractor extracts metrics from C source code and displays them. The Software Metrics Extractor starts with the class `CParser`. The component accepts source code input from an input stream through an “in” pipe port `-CodeInput`. The reference to the stream is set by the method `setInputStream`. The component accepts a parameter to display as the name of the frame.

Given all the details of components, the provider is able to create bundled components that will going to be dispatched to the component broker. The creation of components can be done in the application supplied to component providers. For example, Figure 22a shows the environment used to create the `CParser` component. The environment is designed for creating primitive WebCODS components out of Java executables. Users specify interfaces to the component in terms of ports. The newly created ports are typed and they need to be specified by a name and the corresponding method to set up references. The graphical representation of the component are transformed into textual description and saved into the local environment for later referral by other composite components.

After specifying the primitive components, the provider can use them to create new composite components. Linking the preprocessor and the software metrics extractor together can form a C Software Metrics Extraction System called `CompleteParser`. Components in the system can be linked together with either a Non-Buffered Connector or a Buffered Connector. The composer selected the linkage to be a Non-

Buffered Connection. Figure 22b shows the environment used to generate the system. The instantiation of this component requires two parameters. The first parameter is distributed to the CParser and the Preprocessor accepts the second parameter.

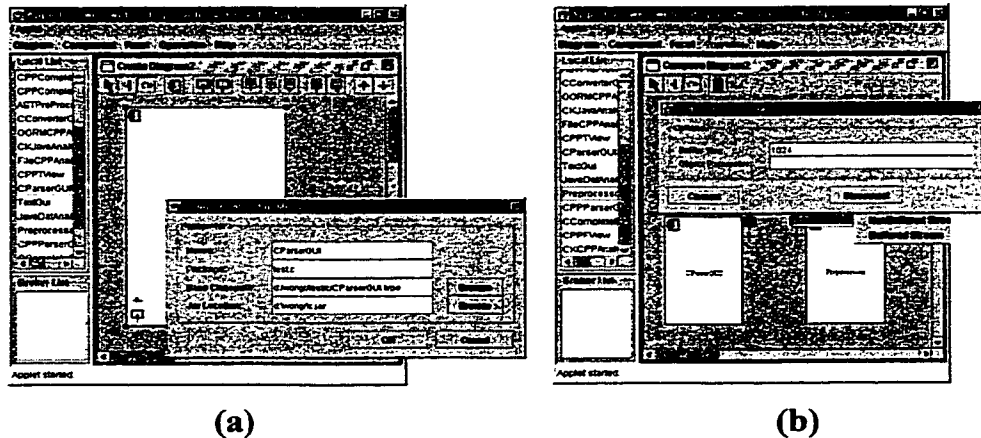


Figure 22: Creation of (a) primitive and (b) composite components in WebCODS

Figure 23 shows the descriptions generated from the environment for CParser, Preprocessor and the CompleteParser system. The provider can transfer these newly created components to the broker. The interpreter in the composition environment converts the textual description of the selected component into OO-based description and determines the bytecode that needs to be transferred. The bytecode is compressed in Jar format and is transferred with the serialized description using a protocol defined in the JavaSpaces.

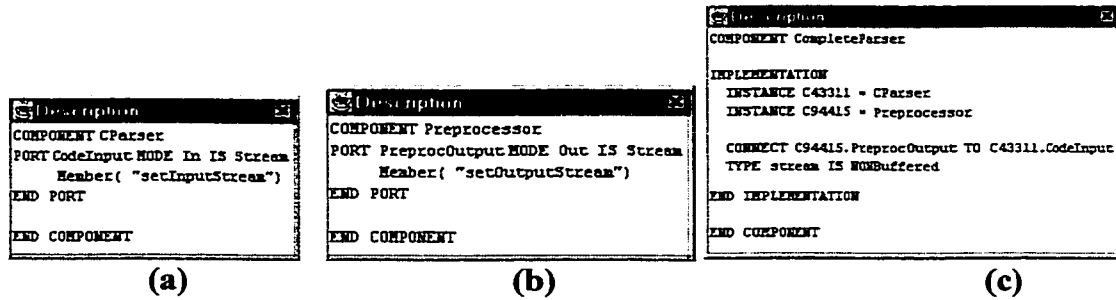


Figure 23: Generated Description for UniCon-based Description in WebCODS for (a) CParser, (b) Preprocessor and (c) CompleteParser

The last step of component creation is to specify component descriptions (Figure 24). The provider is supplied with a dialog box to specify the facets and free-text descriptions. The core facets must be filled with attributes. If the available list of facets is not enough to classify the components, new facets can be added. The free-text description allows providers to describe the component in natural language. All the specified information is used by the search engine and is viewable by clients after downloading the component information.

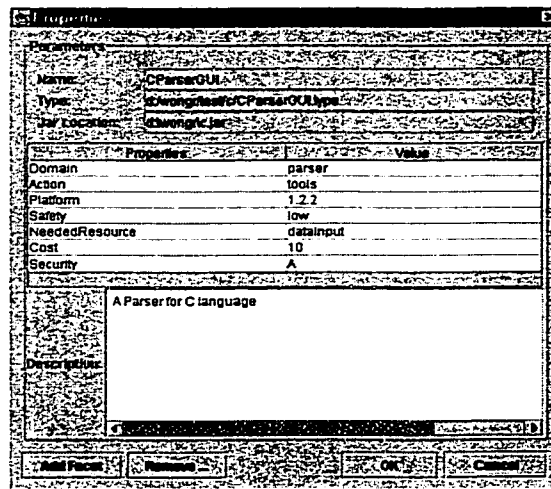


Figure 24: Specifying classification details for components

10.2 Broker

The Broker provides an environment similar to a market where Customers and Providers meet and trade products. The broker is the administrator of the market; it provides a framework for providers to advertise their software products. Customers enter the market to browse and retrieve software components that fit their needs. Customers who are interested in a component can search the broker for matches or pick from a list containing all available components. The broker transfers the required product to the client's environment for composition and execution.

After preparing components in the provider environment, the provider is ready to publish the component `Preprocessor` and `CParserGUI` on the broker. The provider encrypts the component using a DH secret key. The secret key is generated from the private key of the provider and the received public key from the broker. The encrypted components are then signed and sent to the broker. When the broker receives the component, the encrypted components are decrypted and the identity of the sender is verified. If the received component passes the security test, the broker creates an entry for the component in the database. The classification details of components are analyzed and stored in the search engine.

When a client makes a query for components in the broker, the broker generates an XML document with all the information regarding the obtained result from the search engine. This generated XML document will be transferred to the client. The document contains also the classification details of all the listed components. If the client is interested in any of the contained components, the broker will encrypt the component with the secret key shared between the broker and the client. The encrypted component is sent to the client through the security center. The component is sent to the client and can be decrypted with the same secret key.

10.3 Clients

After connecting to the broker, clients can obtain a list of available components or search for matching components from the broker. The query is made using acquisition form that helps the client visualize the name of the facets and permits the choice of terms in a controlled vocabulary. Figure 25 shows that search the broker using the name of providers, attributes in facets, and free-text. When query components using facets, the Properties, the Value and the Bounds columns are used to specify matching parameters. The Properties column shows the facets currently defined in the search engine. The Value column contains the terms that have been used to describe the facet. The Bound column allows searchers to match components using word or numerical comparison. For example, the component searcher can use the numerical comparison to query components using cost facet by specifying the upper and lower bound of the matching requirements. The example shows that the matching is based on the Domain facets. However, new rules can be added to the form by clicking on the “Add Rule” button to add more specific matching criteria.

The searching parameters used for searching are stored in the search engine and are downloaded to the client environment dynamically for each query.

Properties	Value	Bounds
Domain	equal	
parser		
metrics		
programing		
analyze		

Figure 25: Query dialog box

The query returns an XML document to the client. This document contains a list of components in descending order of matches. The document can be visualized as a tree (Figure 26). The client can view all the specified classification details of any components in the list by selecting the attributes. If clients are interested in using any matched components, they can click the “Load” button to download the component from the broker.

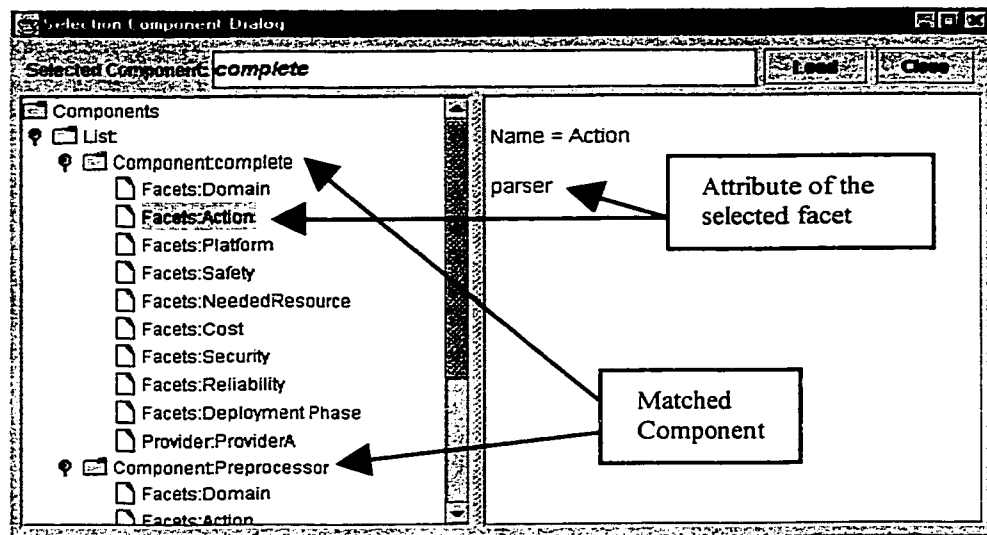


Figure 26: The selection dialog after querying the broker

Clients download components from a selection panel and store the components in their local environment. By downloading the `Preprocessor` and the `CParser` from the broker, clients can build a C Metric Extraction System. In the composition panel, clients link the obtained components with a connector of stream type. Figure 27a shows the resulting graphical component editor panel when the client has downloaded the primitive components and linked them together with a Buffered Stream Connector with a buffer size of 1024 bytes. After the configuration of the connector, the downloaded components are executed and linked together with the parameterized stream connector. Clients can also request the predefined `CompleteParser` component to avoid linking of components (Figure 27b). During the analyzing of component descriptions from unresolved to resolved component, connections

between subcomponents are reestablished and all referred components are executed. The executed components with the specified parameters are shown in Figure 28.

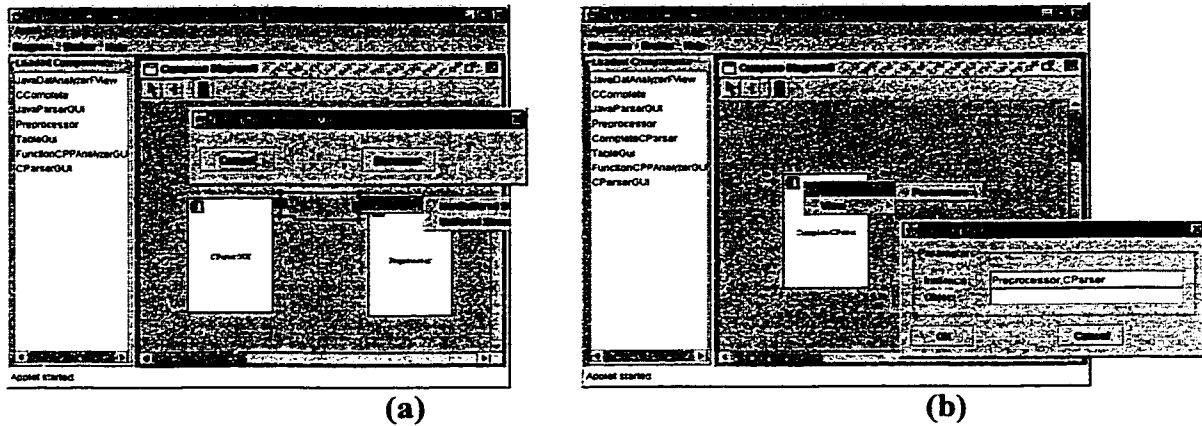


Figure 27: The graphical component editor in client panel used to (a) compose components and (b) standalone component

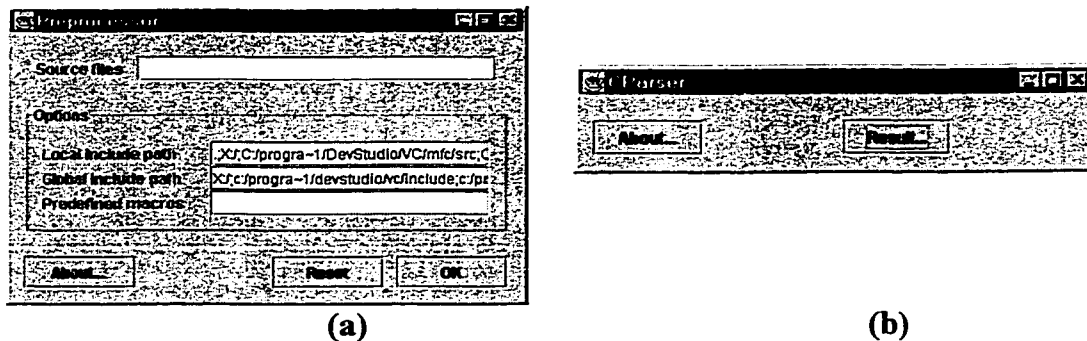


Figure 28: The instantiated components with the parameters specified at runtime (a) Preprocessor and (b) CParser

10.4 Integration of Existing Applications into WebCODS

Existing Java applications can be converted into WebCODS components. The conversion requires a wrapper class to:

1. Characterize the interface of the component
2. Provide interaction points for connectors to communicate to the component
3. Specify the execution sequence of the component

For example, a Java application, `dx2j3d` is a Java freeware utility that converts an object drawn out of 3D Polylines in AutoCad `dx2` file format into a ready-to-compile Java3D java file (Washburn, 1999). The application contains 5 classes bundled in a jar file. The command line used to execute the application is:

```
java dx2j3d <filename.dxf> -<option>
```

The utility reads in the name of the input file from the command line and outputs a .java file that uses the same name as the AutoCad .dxf file. The options available for the application are listed in Table 11 below.

Option	Descriptions
-?	Lists options
-nb	Turn off the use of bubble sort in CAD layers
-nf	Turn off the use of face sort in conversion

Table 11: Options used in the `dx2j3d` component

The interface of the component consists of an input port to read files and an output port to output files. The input port provides the name of the file that will be processed by the utility. The input file name supplied from the connector is saved in the wrapper class and used when the component is executed. After execution, the component outputs the result in the file. The final step is the mapping of the output file to the file specified in the output File connector.

The command line execution of a Java application can be simulated in the environment by invoking the `main` method. The arguments required for execution are supplied at the time of execution.

The wrapper class required for dxf2j3d utility is shown in Figure 29.

```
public class DXF2J3DConverter extends Thread {

    String outputFileName, inputFileName;
    public void setOutputFileName(String name)
        {outputFileName = name;}
    public void setInputFileName(String name)
        {inputFileName = name;}

    String[] arguments;
    public DXF2J3DConverter(String[] args) {arguments = args;}
    public DXF2J3DConverter createInstance(String[] args)
        {return new DXF2J3DConverter (args);}

    public void run() {
        //Create reordered arguments
        String[] newArgument = messageArguments
                                (inputFileName, arguments);

        //Execute the application
        Djf2j3d.main(newArgument);
        //Mapping the current output with the connector
        mappingOutputFile(outputFileName);
    }
    ...
}
```

Figure 29: Wrapper class for dxf2j3d utility

The wrapper class provides two methods for File connectors to set up input and output file names. The names are stored in the class as instance variables, and used when the component executes.

The passing of runtime parameters to the component uses the `createObjectWithParameters` method. The method in this case does not actually create any object instances of `dxf2j3d`; it stores the parameters to be used when the component instantiates.

The wrapper class extends the `Thread` class to avoid deadlock of components in the instantiation environment. Subclasses of the `Thread` class are required to implement the corresponding `run` method. The `run` method defines the steps to execute the application.

According to the specification of the application, it requires the first argument to be the name of the input file followed by any options. The execution of `dx2j3d` utility requires three steps. The first step of execution is the recreation of the arguments by picking up arguments obtained from the input file connector and the instantiation environment. The component can then be executed by invoking the `main` method with the reordered arguments. After execution, the output file from the application is converted to the file specified by the output file connector.

After the wrapper class is developed, it is compiled and bundled into a jar file with all other necessary executables. The component provider can create the corresponding description for the newly formed primitive component and market it as a component to the broker.

The given interface allows other WebCODS components to be connected to the input and output port of `DXF2J3Dconverter`. For example, a file editor can be connected at the input end for editing the file input to the converter, and a viewer at the output end to view the produced java file.

In the composition environment, the user downloads the `DXF2J3Dconverter`, a file editor with a file output port and a viewer with a file input port. After downloading the components, the user performs component composition by linking them together with 2 file connectors. The file connectors are specified with the names of files being input into and output from `DXF2J3Dconverter`. Figure 30a shows the use of file connectors to compose applications with downloaded components.

The DXF2J3Dconverter can be executed with parameters supplied at runtime. Figure 30b shows that the converter is instantiated with the `-nb` option to turn off the use of bubble sort in the CAD layers. After all the parameters are set, the application is ready to be executed.

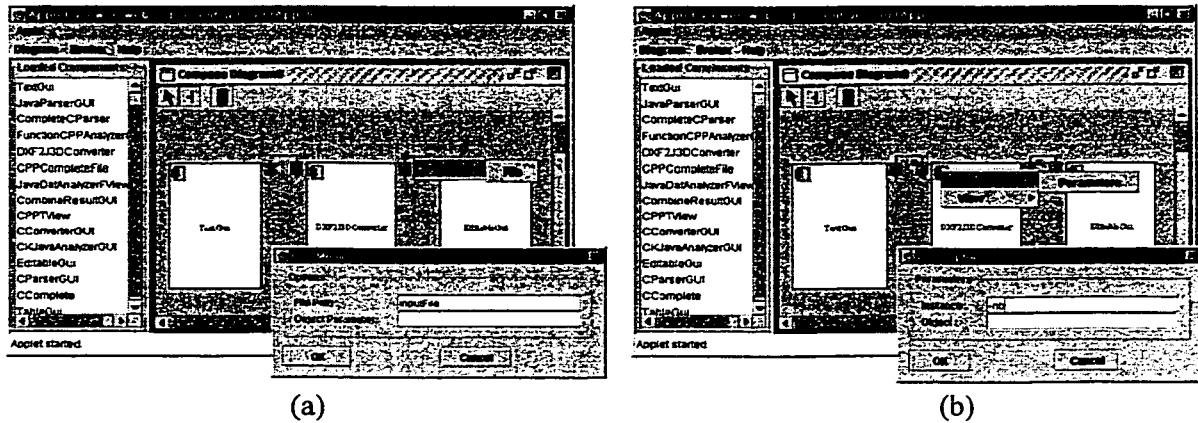
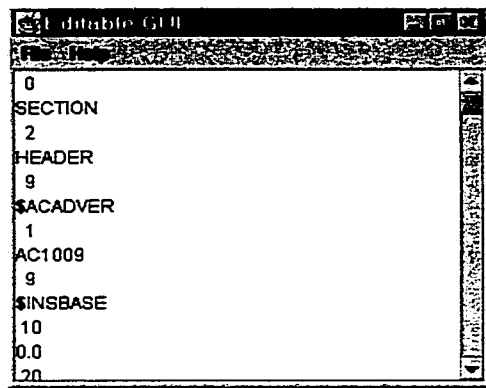
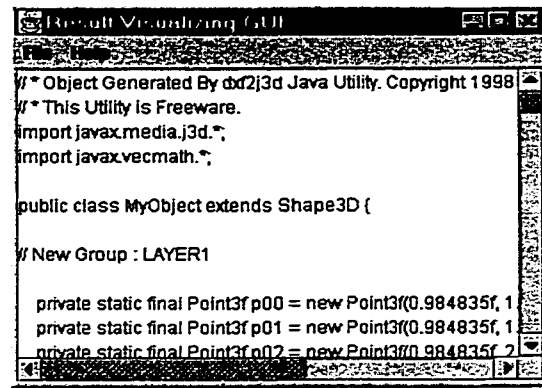


Figure 30: The (a) composition and (b) execution of the DXF2J3Dconverter components in the WebCODS environment

After the execution of the composed application, Figure 31a shows the file editor that sends the file to the DXF2J3Dconverter. The DXF2J3Dconverter processes the received file and outputs to another file specified by the output file connector. Figure 31b shows the file viewer that is able to read the file specified by the connector.



(a)



(b)

Figure 31: The instantiated components resulting from the composed application
(a) file editor and (b) file viewer

Chapter 11: Conclusions and Further Research

This thesis has described the experience in developing a system to support dynamic composition of software components in a Web-based environment.

The system defines the responsibilities of each involved participant: component broker, clients and provider. Providers develop components and offer them to customers through the mediation of the broker. Customers visit the broker and download required software components. They can use the components as they are, or compose them together either with a composition language or a graphical component editor. The future version of the system should contain a charging mechanism based on the usage of the components.

The use of a compositional style to build applications affects how software is produced. WebCODS envisioned two types of developers: (1) component developers and (2) ware developers. Providers are components developers. They can follow either a top-down or bottom-up approach to develop software. Larger applications can be sub-divided into smaller component-wares. Small components are interconnected to yield larger ones. Clients are ware developers following a bottom-up approach. They use pre-packaged components to form applications.

The development of the system is separated into two areas: component composition and component brokerage.

The component composition environment developed in the system focuses on pre-compiled Java components and UniCon-based descriptions. The system supports two types of components: primitive components and composite components. In this way, applications can be composed using existing components to create customizable applications.

Primitive components are the basic building blocks. They keep references to the implementation of the Java components. Each primitive component has an associated list of ports that defines the supported mechanisms for its interconnections.

Composite components are defined as a collection of other (primitive or composite) components, possibly connected through connectors. Composite components keep a description of the instantiation steps that detail how each enclosed component has to be generated and connected to other components. As with primitive components, they have an associated list of ports.

When components are downloaded from the broker to the client's environment, the components can be loaded into the application dynamically. The execution environment for WebCODS components supports a mechanism to construct the component using an extension of the UniCon ADL language. The components are instantiated and executed upon arrival in the clients' composition environments.

The selected implementation language, Java, provides platform independence for components as well as other useful features, such as, introspection, object-serialization and classloading. The dynamism property of the system is supported – components can be loaded into the system at anytime and executed within the composition environment.

Components are composed using a predefined set of connectors. The dynamic property of Java allows the connectors to be encoded in template format. The instantiation of connectors does not require the generation of customized code and recompiled by compiler to obtain binaries. The connection between components is dynamic and instant. The composition environment supports dynamic evolution of software architecture by reconfiguring the connections between components.

WebCODS provides brokerage of software components on the Web. The broker of the system can load and unload components into the component-storage dynamically. The components stored in the broker consist of executables, connectivity descriptions and classification details.

The classification details of components are stored in a search engine. The engine is used when clients want to search for components in the broker. The classification scheme in WebCODS uses facets and free-text descriptions. The querying of components is primarily based on matching of facets and uses free-text matching as secondary searching mechanism. The facet-based classification scheme can be modified by introducing new or removing existing facets. The cost of modification for a facets classification scheme has been shown to be less than that for other classification schemes (Frakes, 1991).

The mobility of components is required in the brokerage environment. The transferring of components in the network is encrypted to target the security needs. WebCODS uses DH key agreements to create a secret key for encrypting and decrypting components – only authorized parties can decrypt the message and obtain the component. The encryption technique used in the current implementation is the DES algorithm, but can be easily upgraded to other algorithms like “BlowFish” or “RAS”.

Future work should focus on several issues:

Components in WebCODS can be new or existing applications as shown in the example. However, more composition mechanisms should be extended to support other complex connectors. As suggested by Deline (1999), other connecting mechanisms, such as explicit procedure calls and shared variables are commonly used to compose components. More research should be performed to investigate how these mechanisms are implemented in the composition environment in WebCODS.

The composition environment analyzes the permissibility of connection based on type-checking and direction of operation. The analysis guarantees the syntactic correctness of the connection. Yakimovich *et. al* (1999) suggest that when components interact with each other, there is a high possibility of mismatches between them. Therefore, the description needs to integrate the syntactic and semantic aspects of the connection.

Moreover, the using of WebCODS to support the infrastructure for component based development environment is still under investigation. The procedure for breaking down a monolithic application to composable components is needed. If the procedure is established, the definition for components in WebCODS is clear.

The classification scheme is only in its first version. After obtaining feedback from providers, the scheme can be improved by revising the facets and the related attributes. When identifying components using the search engine, the engine can be improved with automatic construction and statistical analysis to increase the accuracy for locating components. The proximity mechanism used in the thesaurus requires refinements to reflect the proximity relations between synonyms in the domain. In additions to the adjustment in the thesaurus, further statistical analysis can be performed to investigate the preference of the selector.

The current implementation of WebCODS provides a prototype to experience the possibility of component composition and brokerage in the network environment. To improve the usability of the system in commercial environments, additional tools should be investigated, such as servers providing business logic, to provide payment issues.

References

- Allen R., and D. Garlan, "A Formal Basis for Architectural Connection," *ACM Transactions on Software Engineering and Methodology*, 6(3): 213-249, 1997.
- Biggerstaff T., J. Mitbender, and D. Webster, "Program Understanding and the Concept Assignment Problem", *Communications of ACM*, 37(5): 72-82, 1994.
- Biham E., D. Boneh, and O. Reingold, "Breaking Generalized Diffie-Hellman Modulo a Composite is no Easier than Factoring", *Information Processing Letters (IPL)*, 70: 83-87, 1999.
- Boneh D., and R. Venkatesan, "Hardness of computing the Most Significant Bits of Secret Keys in Diffie-Hellman and Related Schemes", *Proceedings Crypto '96*, 1996
- Broadvision, "Broadvision Overview", <http://www.broadvision.com> ,2001
- Bucci P., S. Edwards, W. Heym, J. Hollingsworth, J. Krone, T. Long, W. Ogden, M. Sitaraman, S. Sreerama, B. Weide, S. Zhupanov, and S. Zwben, "Special Feature: Component-Based Software Using RESOLVE," <http://www.csee.wvu.edu/~resolve/resolve.html>, 1994.
- Clemm G., "The ODIN Specification Language", *Proceedings of the 1st International Workshop on Software Version and Configuration Control*, 1988
- Choudhary V., S. Mehta and A. Chaturvedi, "Experiments with Software Renting," *Workshop on Information Systems Economics (WISE)*, 1998.
- Cooprider L. W., *The Representation of Families of Software Systems*, Ph.D. thesis, Carnegie-Mellon University, Computer Science Department, April 1979

- Creech L., F. Freeze, and L. Griss, "Using Hypertext In Selecting Reusable Software Components", *Proceedings of the Third Annual ACM Conference on Hypertext*, 1991
- Ciancarini P., R. Tolksdorf, F. Vitali, D. Rossi, and A. Knoche, "Coordinating Multiagent Applications on the WWW: A Reference Architecture", *IEEE Transactions on Software Engineering*, 24(5)362-375, 1998
- D'Souza D., and A. Wills, *Component-Based Development Using Catalysis*, Addison-Wesley, 1998.
- Damiani E., M. Fugini, and C. Bellettini, "A Hierarchy-Aware Approach to Faceted Classification of Object-Oriented Components", *ACM Transaction on Software Engineering and Methodology*, 8(4): 425-472, 1999
- Dashofy E, N. Medvidovic, and N. Taylor, "Using Off-The-Shelf Middleware to Implement Connectors in Distributed Software Architectures," *Proceedings of the 21st International Conference on Software Engineering*, 1999
- DeLine R., "A Catalogue Technique for Resolving Packaging Mismatch," *Proceedings of the Fifth ACM Symposium on Software Reusability*, 1999
- DeLine R., "Avoiding Packaging Mismatch with Flexible Packaging," *Proceedings of the 21st International Conference on Software Engineering*, 1999
- DeRemer F. and H. Kron, "Programming-in-the-Large Versus Programming-in-the-Small," *IEEE Transactions on Software Engineering*, 2(2): 80-86, 1976
- Devanbu P., Y. Chen, E. Gansner, H. Muller and J. Martin, "CHIME: Customizable Hyperlink Insertion and Maintenance Engine for Software Engineering Environment," *Proceedings of the 21st International Conference on Software Engineering*, 1999
- Diffie W. and M. Hellman, "New Direction in Cryptography", *IEEE Transaction in Information Theory*, 22(6): 644-654, 1976
- Emmerich W., *Engineering Distributed Object*, John Wiley & Sons, 2000.

- Etzkorn L., and C. Davis, "Automatically Identifying Reusable OO Legacy Code", *IEEE Computer*, 30(10): 66-71, 1997
- Faison T., "Interactive Component-Based Software Development with Espresso," *Proceedings of the 1997 International Conference on Automated Software Engineering*, 1997
- Fiadeiro J., and T. Maibaum, "Interconnecting Formalisms: Supporting Modularity, Reuse and Incrementality," *Proceedings of the third symposium on The foundations of software engineering*, 1995
- Frakes W., and T. Pole, "An Empirical Study of Representation Methods for Reusable Software Components", *IEEE Transaction on Software Engineer*, 20(8) 617-630, 1994
- Frakes W., and B. Nejme, "An Information System for Software Reuse", *20th International Conference for System Science*, 1987
- Freeman E., S. Hupfer and K. Arnold, *JavaSpacesTM Principles, Patterns, and Practice*, Addison-Wesley, 1999
- Gamma E., R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional Computing Series, Addison-Wesley, 1994.
- Gelernter D., "Generative Communication in Linda", *ACM Transactions on Programming Languages and Systems*, 7(1): 80-112, 1985
- Goguen J., "A Categorical Manifesto", *Mathematical Structures in Computer Science* 1(1): 46-67, 1991
- Gosling, J., and H. McGilton. "The Java Language Environment - A White Paper." Javasoft, May 1996. <http://java.sun.com/docs/white/index.html>
- Gupta A., C. Ferris, Y. Wilson, and K. Venkatasubramanian. "Implementing Java Computing: Sun on Architecture and Applications Deployment." *IEEE Internet Computing*, 2(2): 60-64, 1998

- Henninger S., "An Evolutionary Approach to Constructing Effective Software Reuse Repositories", *ACM Transactions of Software Engineering Methodologies*, 6(2): 111-140
- Hewlett-Packard, "A Tutorial in Developing e-speak Services," <http://www.e-speak.net/library/pdfs/tutorial.pdf>, 2000
- Hoare C., *Communicating Sequential Processes*, Prentice Hall, 1985.
- Hummel, R. "How Java Can Pay the Rent." Byte Magazine, June 1996
- Lam S., and A. Shankar, "A Theory of Interfaces and Modules I – Composition Theorem," *IEEE Transactions on Software Engineering*, 20(1): 55-71, 1994.
- Li K., G. Lifeng, M. Hong, F. Yang, "An Overview of JB (JadeBird) Component Library System JBCL", *Proceedings of The Technology of Object-Oriented Language and System Tools*, 1998
- Kemmerer R., "Security Issues in Distributed Software", *Proceeding of the 6th European Conference on Software Engineering*, 1997
- Kent S., K. Lano, J. Bicarregui, A. Hamie, and J. Howse, "Component Composition in Business and System Modeling," *Proceedings of OOPSLA97 Workshop on Object-Oriented Behavioral Semantics*, 1997
- Magee J., and J. Kramer, "Dynamic Structure in Software Architectures," *Proceedings of the 4th Symposium Foundations of Software Engineering*, 1996.
- Magee J., J. Kramer and M. Sloman, "Constructing Distributed Systems in Conic," *IEEE Transactions on Software Engineering*, 15(6): 663-675, 1989.
- Medvidovic N., and R. Taylor, "A Classification and Comparison Framework for Software Architecture Description Languages," *IEEE Transactions on Software Engineering*, 26(1): 70-93, 2000.
- Medvidovic N., D. Rosenblum, and R. Taylor, "A Language and Environment for Architecture-Based Software Development and Evolution", *Proceedings of the 21st International Conference on Software engineering*, 2000

- Mozer M., "inductive Information Retrieval Using Parallel Distributed Computation", ICS Rep. 8406, Institution for Cognitive Science, University of California — San Diego, 1984
- Michiels B., and B. Wydaeghe, "Component Composition", *Proceedings of the 22nd International Conference on Software engineering*, 2000
- Milner R., J. Parrrpw, and D. Walker, "A calculus of mobile processes, Part I and II," *Journal of Information and Computation*, 100: 1-40, 41-77, 1992
- Nierstrasz O., and T. Meijler, "Research Direction in Software Composition," *ACM Computing Surveys*, 27(2): 263-264, 1995
- Ousterhout J., "Scripting: Higher Level Programming for the 21st Century", *IEEE Computer Magazine*, March 1998.
- Paradigm Plus, "PLATINUM technology - Manage and Improve your IT Infrastructure", <http://www.platinum.com>, 2001
- Pozewaunig H., and D. Reithmayer, "Support of Semantics Recovery during Code Scavenging uisng Repository Classification", *Proceedings of the Fifth Symposiun on Software Reusability*, 1999
- Prieto-Diaz R., "A Software Classification Scheme", Ph.D. dissertation, University of California, Irvine, CA, USA, 1985
- Prieto-Diaz R., and J. Neighbors, Module Interconnection Languages, *The Journal of System and Software*, 6(2): 307-334, 1986
- Prieto-Diaz R., "Implementing Faceted Classification for Software Reuse", *Communications of ACM*, 34(5): 88-97, 1991
- Prieto-Diaz R., and P. Freeman, "Classification Software for reusability", *IEEE Transactions on Software Engineering*, 4(1): 6-16, 1987.
- Puliafito A., O. Tomarchio, and L. Vita, "Porting SHARPE on the WEB: Design and Implementation of a Network Computing Platform using Java", *9th IEEE International Conference on Modeling Techniques and Tools*, 1997

- Rational Rose, "Rational Rose v2001: Visual Modeling, UML, Object-Oriented, Component-Based Development with Rational Rose", <http://www.rational.com/products/rose/index.jsp>, 2001
- Sametinger J., *Software Engineering with Reusable Components*, Springer 1997
- Scott O., *Java Security*, O'Reilly, 1998
- Shaw M., "Research Opportunities in the Virtual Agora: Market Aspects of Open Resource Coalitions," *First International Workshop on Economics-Driven Software Engineering Research*, http://www.cs.cmu.edu/afs/cs.cmu.edu/project/vit/www/paper_abstracts/Virt_Agora.html, 1999.
- Shaw M., and P. Clements, "A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems," *Joint Proceedings of the 2nd International Software Architecture Workshop and International Workshop on Multiple Perspectives in Software Development (SIGSOFT '96)*, 1996.
- Shaw M., R. DeLine, and G. Zelesnik, "Abstractions and Implementations for Architectural Connectors," *Proceedings of the Third International Conference on Configurable Distributed Systems*, 1996.
- Shaw M., R. DeLine, D. Klein, T. Ross, D. Young and G. Zelesnik, "Abstractions for Software Architecture and Tools to Support Them," *IEEE Transactions on Software Engineering*, 21(4): 314-335, 1995.
- Sitaraman M., "Why Neither Java Components Nor Formal Methods Can Do it Alone," *Proceedings of the 9th Workshop on Institutionalizing Software Reuse*, <http://www.csee.wvu.edu/~resolve/papers/wisr-paper.html>, 1999.
- Succi G., C. Bonamico, L. Benedicenti, E. Liu, T. Vernazza and R. Wong, "Supporting Electronic Commerce of Software Products through Pay-Per-Use Rental of Downloadable Tools", *Internet Commerce and Software Agents: Cases, Technologies and Opportunities*, Idea Group Publishing, 2000.
- Sun Microsystems, "JavaSpaces[™] Specification", July 1998
<http://java.sun.com/products/javaspaces/specs/js.pdf>

- Sun Microsystems, “*Java™ 2 Platform, Standard Edition Documentation (J2SETM)*”, 1999 <http://java.sun.com/docs/index.html>
- Swanson J., and M. Samadzadeh, “A Reusable Software Catalog Interface”, *Proceedings of the 1992 ACM/SIGAPP Symposium on Applied computing*, 1992
- Szyperski C., *Component Software - Beyond Object-Oriented Programming*, Addison Wesley, 1998.
- Tanenbaum, H. Stavaren, E. Keizer, and J. Stevenson, “A Practical Tool Kit for Making Portable Compilers” *Communication of ACM*, 26(9): 654-662, 1983.
- Thomas J., *Module Interconnection in Programming Systems Supporting Abstraction*, Ph.D. thesis, Brown University, June 1976.
- Weinreich R., “A Component Framework for Direct-Manipulation Editors,” *Proceedings of the Technology of Object-Oriented Languages and Systems*, 1998
- Williams M., “What makes RABBIT run?”, *International Journal Mechanic Machine Studies* 21, 333-352, 1984
- Yakimovich D., J. Bieman and V. Basili, “Software Architecture Classification for Estimating the Cost of COTS Integration”, *Proceedings of the 21st International Conference on Software Engineering*, 1999.
- Yourdon E., “Java, the Web and Software Development.” *IEEE Computer*, 29(8): 17-34, 1996

Appendix A: WebCODS User Manual

The execution of WebCODS requires the following software packages.

1. Java 1.3 Standard Edition
2. Jini 1.0.1
3. Jini Extension kit (JXE) 1.0.1
4. Java Cryptographic Extension Kit (JCE) 1.2
5. XML Parser from IBM
6. Graphical Editor Framework (GEF)

The hardware requirements:

- Pentium II or higher
- 20M of free spaces

The WebCODS system consists of three main executables:

1. Broker – used by the administrator of WebCODS
2. Provider – used by the parties supplying components to the Broker
3. Client – used by the parties downloading components from the Broker

A.1 Broker

The execution of the broker requires the following to be running at the same time.

1. HTTP Server – The server is provided for client and provider programs to download the necessary classes for execution. The server can also be used to supply other software packages, such as JCE, XML Parser and GEF. The use of the server does not require executing the client nor does the provider program need to have all the software packages installed.
2. RMI Daemon (RMID) – The daemon is used by by the Jini Lookup Service. The daemon is an extension to the RMI registry. The extension allows the registry to handle persistence storage for remote objects. The persistence information of the registry can be logged in a file specified by the user.
3. Jini Lookup Service – The Lookup service provides facilities for services to advertise their availability and for would-be clients to obtain references to those services. In WebCODS, the Jini Lookup Service is used to locate the JavaSpaces service available in the Jini server.
4. JavaSpaces Server – The medium used for transferring components in WebCODS. The JavaSpaces is implemented as a service of Jini. Different names are used to distinguish different services of JavaSpaces available in the Jini Lookup Service.

The HTTP server can be any commercial server. The RMI Daemon is a standard program that comes with the Java Development Kit. The RMID program is tied to specific Java versions, meaning that RMID will only work with its particular version. The execution of the Jini Lookup Service and JavaSpaces Server requires the installation of Jini and Jini Extension Kit. The detailed procedures for executing both services are discussed in the documentation that comes with the Jini package.

The execution of the broker depends on several data files:

1. Database file
2. INI file
3. Keystore file
4. Facets file
5. Thesaurus file

These files must be located at the same directory that executes the broker program.

A.1.1 Database File

The database contains all the clients' and providers' information. The current database is built with Microsoft Access. The database contains tables to store users' login name, password and the company that they belong to. The location and name of the database is installed to the system's registry with a script specific for the Windows NT operation system. The installation of the database into the registry allows the ODBC service from the operating system to gain access to the file.

Accessing the database is based on a JDBC bridge. The bridge allows the broker program to link to the ODBC bridge of the operation system. By specifying the name of the database, the program gains access to the database installed in the local system. The communication between the program and the database is based on SQL (Structured Query Language), the standard language for accessing relational databases.

With the correct JDBC and ODBC bridges, there is no limitation on the kind of database that is used in WebCODS.

A.1.2 INI File

The INI file stores the necessary information that is used to start up the server. The file contains 4 pieces of information:

1. Name of the database – name of the database stored in the registry of the system
2. JDBC bridge used with the database – the correct JDBC bridge to be used with the database
3. Location of a Jini Lookup Service – the IP address of the machine that runs a lookup service
4. Name of the JavaSpaces – the name of the JavaSpaces that will be used by WebCODS as transferring medium. The instance of the JavaSpaces service must exist in the specified machine that runs the Jini Lookup Services.

The information provided in the INI file will be input to the broker when the broker program starts to execute.

A.1.3 Keystore File

The keystore file stores the certificates of the broker itself and the certificates of clients. The manipulation of the keystore requires the use of the keytool available with the JDK. The tool can be used to add and remove certificates from the keystore file. Moreover, the keytool program is able to generate and store a key pair – private and public key. The private key will be stored in the keystore, and the public key can be retrieved. The public key can be sent to certificate authorities for verifying and obtaining a verified certificate.

Certificates obtained from the certificate authorities can be installed into the keystore and becomes a trusted certificate. This type of entry contains a single public key certificate belonging to another party. It is called a trusted certificate because the keystore owner trusts that the public key in the certificate indeed belongs to the identity identified by the owner of the certificate. More complete information about the keytool program can be found at:

<http://developer.java.sun.com/developer/onlineTraining/Security>

When the WebCODS administrator runs the broker for the first time, the administrator needs to set up the keystore file. The keytool is used to generate a set of private and public keys for the broker. The public key is then sent to a certificate authority for verification. After the verification, the returned certificate is installed to the keystore. The broker program is able to access its own public and private keys.

For all registered clients and providers in the application, their public key must be added to the keystore. The installed public keys and certificates are used to verify the identification parties participating in WebCODS.

A.1.4Facets File

The facets file contains the core facets that are used in classification. The file is read when the search engine is instantiated in the broker. The presentation of the facets is in descending order according to their importance in the classification scheme.

When clients query the broker, the search engine presents the ordered set of facets that is contained in the file.

Currently the facet file contains 6 entries:

- **Domain:** The names of domain that the component is used or may be used. The term “domain” characterizes a set of similar systems or applications that share some common functionality.
- **Action:** The function that is performed by the component.
- **Platform:** The software platform that is used to execute the component.
- **Safety:** The safety level of executing the asset.
- **Needed Resource:** Other related sources required by the successful execution of the component.
- **Cost:** The cost of using the component charged by the broker.

The facets list in WebCODS is not statically built in the search engine and other related components. When the facets list is loaded into the search engine, any components that use the list will query the search engine to obtain the concurrent facets list. Since the facet list is dynamically loaded into the framework, any operations to the list can be modified without affecting other components in WebCODS.

A.1.5 Thesaurus File

Each facet may be accompanied with a thesaurus file. The thesaurus contains a list of synonyms used in the corresponding facets. For each synonym contained in the thesaurus, there is a ranking of how close it is related to each other. The current broker version includes 3 sample thesaurus files:

1. Action.thesaurus – Thesaurus for Action facets
2. Description.thesaurus – Thesaurus for Description facets
3. Domain.thesaurus – Thesaurus for Domain facets

In the thesaurus file, each line contains a set of synonyms and their weighted relations. For example, the term “metrics collecting” and “parsing” are equivalent and the assigned weight for their closeness value is 1.0. The line that appears in the thesaurus file will be:

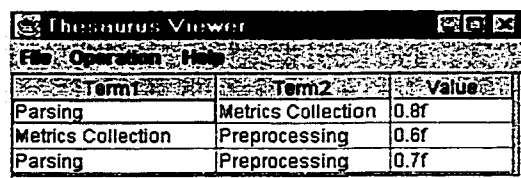
```
metrics collecting, parsing, 1.0
```

A collection of the related synonyms and their associated weights will build up the thesaurus required in the search engine. WebCODS provides a tool, called Thesaurus Viewer, for the administrator to build up the thesaurus. The command to run the class is given below:

```
java webcods.engine.ThesaurusViewer
```

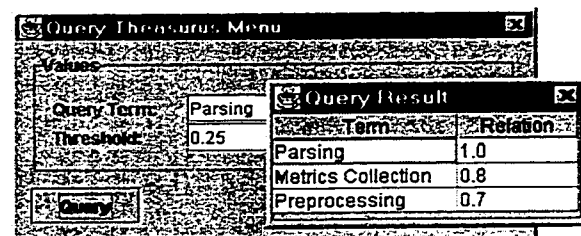
The tool is capable of helping administrators to build and test the thesaurus file. The features of the tool are listed below:

1. Load a thesaurus file specified by the user (Figure 32a)
2. Add and remove entries in the thesaurus file
3. Perform test queries in the build thesaurus (Figure 32b)
4. Save the build thesaurus to be used by the search engine



Term1	Term2	Value
Parsing	Metrics Collection	0.8f
Metrics Collection	Preprocessing	0.6f
Parsing	Preprocessing	0.7f

(a)



Query Thesaurus Menu	
Query Term:	Parsing
Threshold:	0.25
<input type="button" value="Query"/>	

Query Result	
Term	Relation
Parsing	1.0
Metrics Collection	0.8
Preprocessing	0.7

(b)

Figure 32: The tool to manipulate thesaurus (a) loading (b) testing of thesaurus

A.1.6 Execution of the Broker

The main class to execute the broker is `webcods.server.Server`. The successful execution of the broker requires the proper installation of all the software packages specified in Section 1.

The broker program picks up all required input files from the directory that it executes in. It is not necessary to provide any parameters to run the program. If the program fails to start, the corresponding error messages will be displayed on the screen. The administrator of the program needs to fix all the errors before the broker program can execute successfully.

Figure 33 shows the successful execution of the broker in a Windows NT environment. The display shows that the broker has located the Jini Lookup service and matched an instance of JavaSpaces from it. After the initialization, the broker is ready to receive requests for connections from clients and providers.

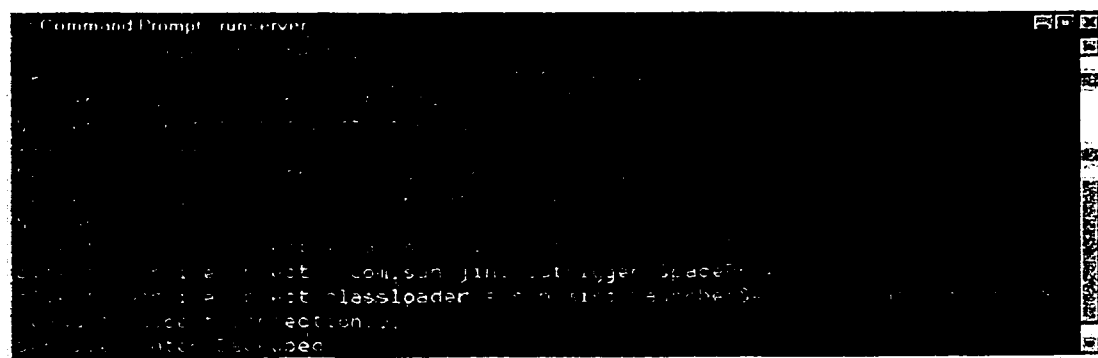


Figure 33: Execution of the broker program

A.1.7 Recovery Service

The broker program is responsible for all the component brokerage related activities. To provide error recovery capabilities, the broker implements a backup mechanism for restoring the state of the broker before a has failure occurred. The backup service is activated every 5 minutes. The backup data is stored in a file called `backup.save`.

When the broker program starts, the recovering service is activated to check if the `backup.save` file exists. If the file exists, the mechanism will recover the last saved status from the file. If the file does not exist, a new broker without any assets will be created.

A.2 Client

The client program is implemented as a Java applet. The application is intended to run with an AppletViewer or any Java-enabled Web-browsers.

A.2.1 Prerequisite to Execute the Application

The execution of the client requires the proper setup of the Java security policies for the local Java Virtual Machine. The additional resources required by the client include: file access, opening network connections, defining new packages in the applet and others. To grant privileges to perform these operations, the boundary of the sandbox in the Java Virtual Machine is redefined with a user-defined policy file. Since the executed binaries are coming from the trusted broker, the sandbox is defined to grant access to all local resources. The policy file contains the following lines:

```
grant {  
    permission java.security.AllPermission;  
};
```

The successful execution of the client application also relies on the software packages described in Section 1. The client has a choice to either install the packages in the local system or download them from the HTTP server hosted by the broker. If clients choose to download software packages from the server, they have to setup the code base for the applet to download the necessary packages. The HTML page is also used to pass the location of the broker to the application. The parameter passing is achieved using a parameter tag. The tag specifies the name of the parameter is “broker” and its value indicates the location of the broker.

The main class used to execute the application is `webcods.client.ClientAppUI`. The HTML page used to execute the client application is shown in Figure 34.

```
<HTML>
<HEAD>
<TITLE>Client Page</TITLE>
</HEAD>
<BODY>

<Applet code=webcods.client.ui.ClientAppUI
        codebase=" titus.quase.ualberta.ca"
        width="598" height="432">
<param name = "broker" value = "titus.quase.ualberta.ca">

</Applet>

</BODY>
</HTML>
```

Figure 34: Sample HTML to execute the client application

The client application also uses a keystore file to store the corresponding public and private key information. Before the WebCODS client runs the application for the first time, the client needs to generate a set of private and public key and submit the public key to the broker for verification. After the verification, the broker will return the verified certificate and the broker's certificate to the client. The client then needs to install both received certificates to its keystore. The certificates are used to perform digital signing, encrypting, and decrypting messages that pass between the client and the broker.

A.2.2 Execution of the Application

Executing the application results in a login screen for the client to connect to the broker. The login process requires the client to enter the user name, the password and the company name assigned by the WebCODS administrator. Figure 35a shows the login screen from the application. After the broker verifies the identity of the client, the application (Figure 35b) shows a list of downloaded components and a panel to perform component composition.

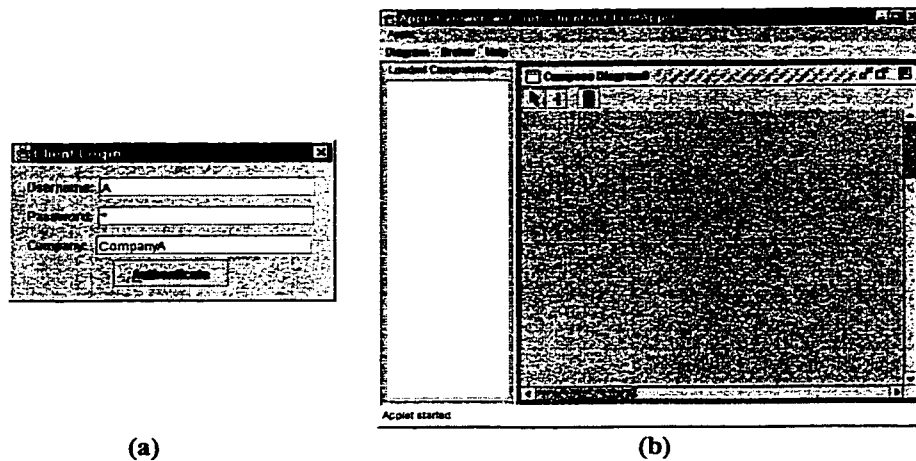


Figure 35: (a) Login screen and (b) environment for the client application

After login to the broker, the client can use the application to query the broker for components. There are two ways of querying (Figure 36a):

1. List all available components in the broker
2. Search the broker using facets or free-text descriptions (Figure 36b)

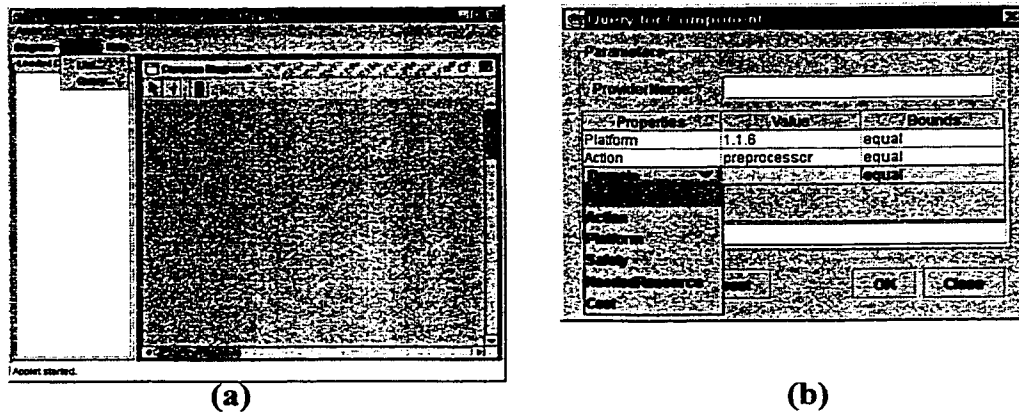


Figure 36: (a) Querying modes in the client environment and (b) searching components in the broker

After listing or searching components available in the broker, a matched list is returned to the client application (Figure 37). The classification details for components is stored in an XML format. A XML viewer is used in the environment so the client can view any attributes contained in the description. If the client is interested in downloading any listed components, the client can select the component in the list and click the load button. After receiving the component from the broker, the component is listed under downloaded components in the main application panel.

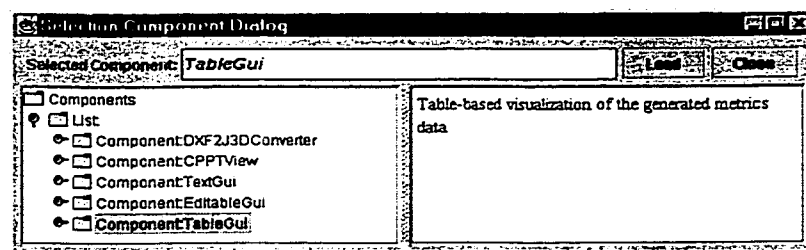


Figure 37: Listing of matched components from the broker

After downloading components from the broker, the client is able to compose applications. Components can be moved from the downloaded component list to the composition panel. The components in the panel are represented as boxes with different icons as ports to signal their availability for further connections. To compose applications, lines are drawn across ports to connect components together. (Figure 38)

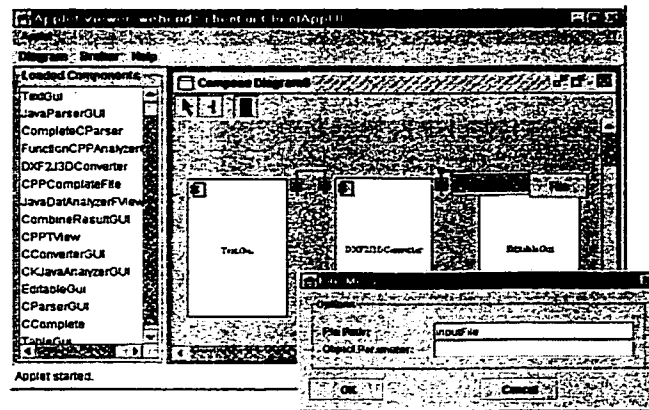



Figure 38: Composition of components in the client environment

The composed application can be executed by clicking on the  icon. The composed application can be modified by adding new components to the structure or re-linking components together in different ways. After the modification of the structure is finished, the client can click the execution icon to run the newly created application.

A.2.3 Saving and Reloading of Components

The composed application can be saved for later retrieval. Clients are allowed to specify the name of the composed component and the location for storing the description. The description of components is generated automatically and reflects the current architecture of applications in the composition panel. Figure 39 shows the saving capability in the client application.

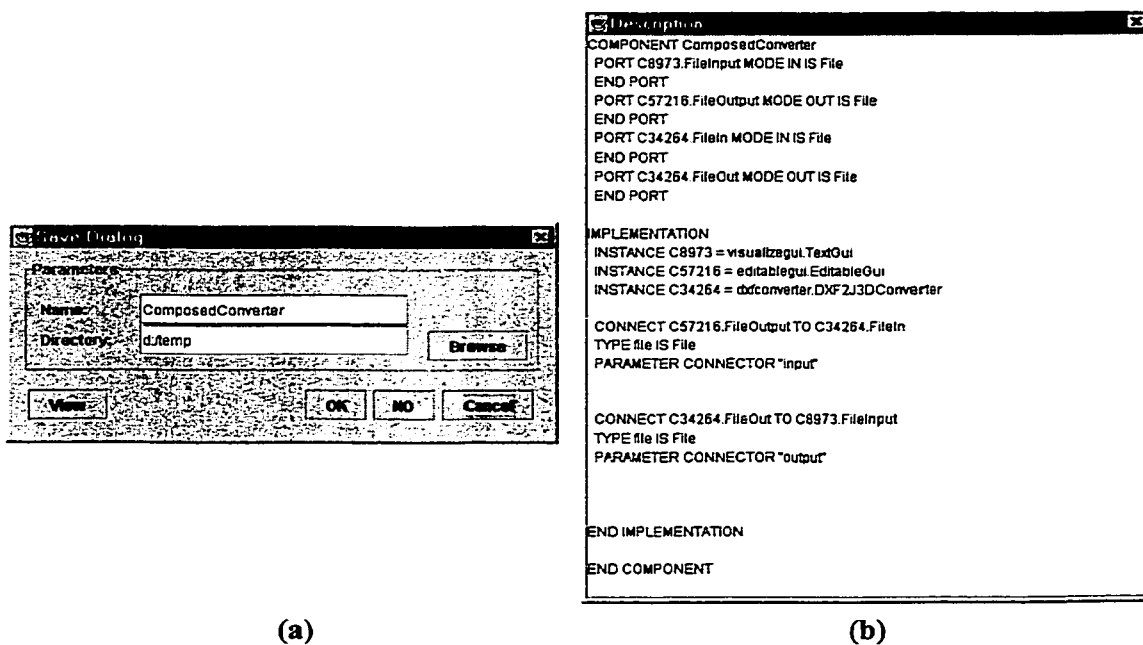


Figure 39: (a) Saving components and (b) the generated description based of the composed application

The saved components can be loaded back to the application as a component. The client uses a file browser to locate the saved components. After the previously saved component is selected and loads into the application, the component is listed under the downloaded components. Saved components in the same way as components downloaded from the broker. Figure 40 shows the reloading of the components back to the client application.

When the client loads the components saved in the local environment, any components referred to in the saved components must be available in the downloaded components list. If any referred component is not available, the component fails to load into the system. If the saved component fails to load because of missing referred components, an error message is displayed to indicate what components are required in the environment. The client should download all the missing components from the broker to ensure successful loading of the saved component. Figure 41 shows the error message when a client loads the saved description without downloading all the required components.

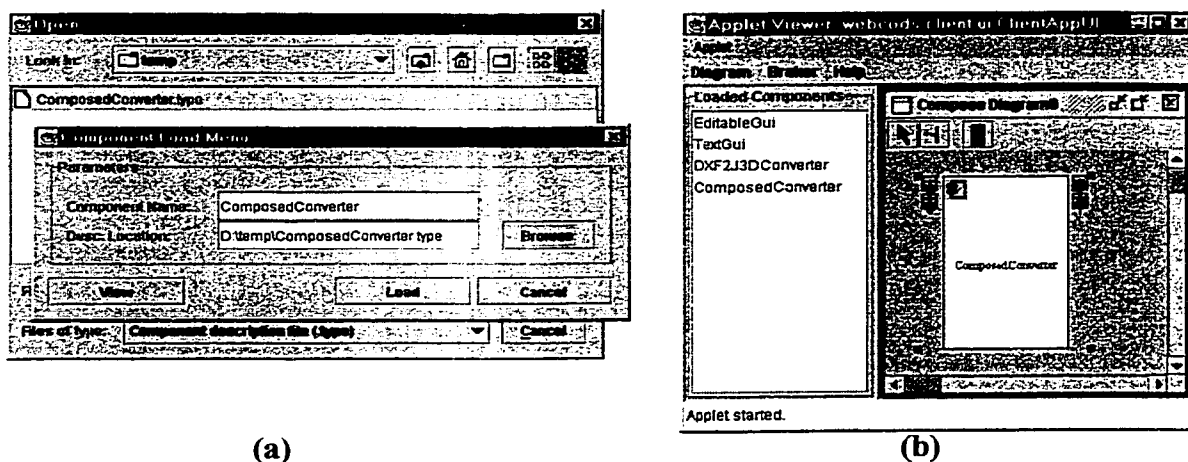


Figure 40: Loading of saved components into the environment

(a) loading and (b) using the component



Figure 41: The resulted error message showing the required components when loading saved description

A.3 Provider

The provider program is implemented as a Java applet. The application is intended to run with an AppletViewer or any Java-enabled Web-browsers.

A.3.1 Prerequisite to Execute the Application

The provider program is used to create and publish components to the broker. The component published to the broker contains 3 pieces of information:

1. Binary Code – executables for the component
2. Classification Information – classification details to be used by clients to identify the component in the broker
3. Connectivity Information – details about the internal structure and further connectivity of the component

The primary components referred to by the application must be installed in the environment. The binary code of these components is contained in separate jar files. These jar files can be created with the jar tool supplied with the standard Java Development Kit. They can be stored in any name and location specified by the provider. When primitive components are created in the application, the classification and connectivity details are stored in the same location as the main executable class.

When composite components are created in the application, the descriptions can be saved in any name and location in the local system. The provider does not need to locate all necessary source code and put them together in a single jar file. The

application will automatically to organize all the required executables when the composite component is published to the broker.

Information about all previously created components in the application is stored in an XML file called `webcods.xml`. Each component saved in the file contains the following information:

1. Name of component
2. Location of the description file for the component
3. Location of the jar file containing the necessary executables (exist only if the component is a composite component)

The execution of the provider application is similar to the running of the client application. The Java Virtual Machine running the application must grant enough privileges to access the local resources.

The HTML page used to execute the client application is shown in Figure 42. The main class used to execute the application is:

`webcods.client.ProviderAppUI.`

```
<HTML>
<HEAD>
<TITLE>Provider Page</TITLE>
</HEAD>
<BODY>

<Applet code=webcods.client.ui.ProviderAppUI
        codebase=" titus.quase.ualberta.ca"
        width="598" height="432">
<param name = "broker" value = "titus.quase.ualberta.ca">

</Applet>

</BODY>
</HTML>
```

Figure 42: Sample HTML to execute the provider application

The provider also has an option of not installing all the software packages described in Section 1. By specifying the code base tag in the HTML code, the application is able to download them from the HTTP server. Another parameter in the code is the “broker”; it is used to indicate the location of the broker.

The provider application also uses a keystore file to store the corresponding public and private key information. The procedure for setting up the file is the same as the WebCODS client.

A.3.2 Execution of the Application

Executing the application results in a login screen for the provider to connect to the broker. The login process requires the client to enter the user name and password assigned by the WebCODS administrator. Figure 43a shows the login screen for the application. After the broker verifies the identity of the broker, the application (Figure 43b) shows a list of previously created components, the submitted components of the provider in the broker and a panel to create components.

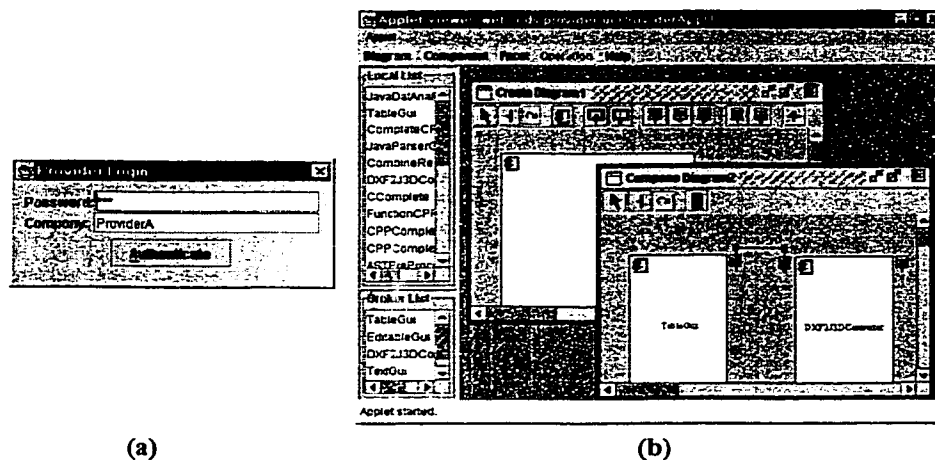


Figure 43: (a) Login screen and (b) running environment for the client application

The provider can publish any components available in the local list to the broker. The provider can select any components listed under local list and display a popup menu for component operation by right clicking on the mouse button. This menu allows providers to view the connectivity and classification description of the component and also contains an option to publish components to the broker. When submitting components to the broker, the application obtains the binaries from the corresponding jar files and the required descriptions from the local environment. After successfully

publishing the component to the broker, the component is listed under the list of components in the broker.

A.3.3 Creation of Components

The application can be used to create primitive and composite components. The primitive components are created using the create diagram in the application. The creation of composite components is similar to composing applications in the client environment.

Composite components are created using components in the local component list. After composing the application, the provider needs to specify classification information for the component and save the description in the local environment. The provider does not need to bundle the required executables into a single jar file, the system can automatically find out the required primitive components and combine all the executables together when the provider submits the component to the broker.

When creating primitive components in the application, the provider needs to specify the following information (Figure 44):

1. Name of the component
2. Package name of the component
3. Path to the component package
4. Location of the jarred executables

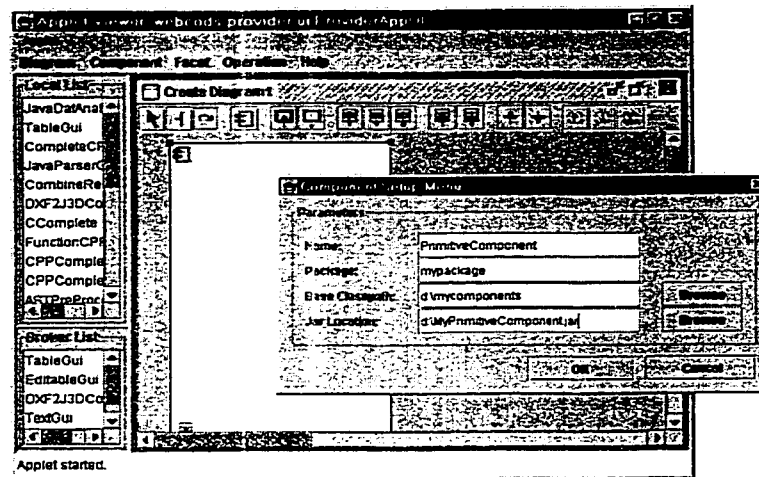


Figure 44: Creation of primitive component

After specifying the necessary information to execute the component, the provider is required to add connectivity details. The available ports are shown in the menu bar of the panel. When providers want to add a port to the component, they can pick the desired one and drop it onto the component. Table 12 lists the icons and their represented ports type.

Port	Direction		
	<i>In</i>	<i>Out</i>	<i>In/Out</i>
Stream			N/A
File			N/A
Event			N/A
Socket			
Datagram Socket			

Table 12: Icons for ports in WebCODS

After dropping the ports onto the component, the provider needs to set up the name for the port and the corresponding method in the executables that is used when connecting the port using a connector.

Figure 45 shows the specification of a file port for a component. The name of the port is called FileInput and the method associated with the port is setInputFile. Once all the ports are setup for components, the connectivity description of the component is created and saved in the same location as the main executable class.

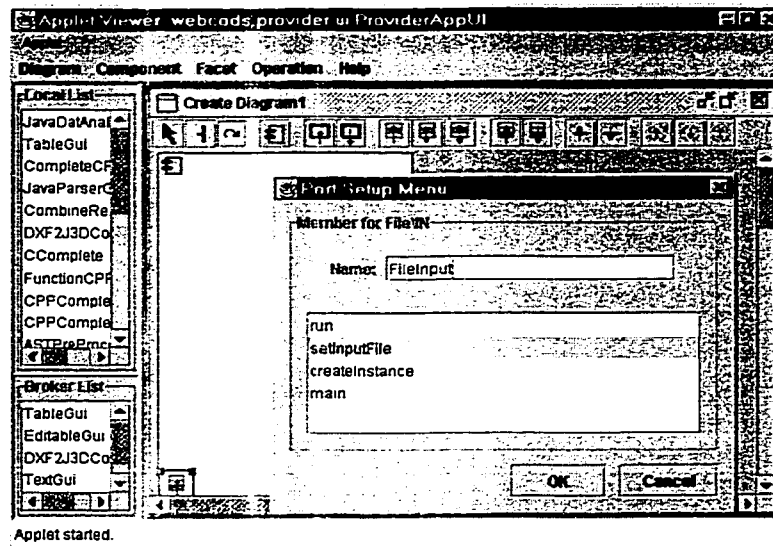


Figure 45: Specifying details for ports using the create diagram

A.3.4 Classification of Components

The final process in the creation of new components is to specify the classification details (Figure 24). The classification details include facets and free-text descriptions.

After specifying connectivity information, the application supplies the provider a dialog box to specify facets and free-text descriptions. The core facets must be filled with attributes. The attributes can be selected from a supplied list or the provider can input a new term to describe the facet. If the available list of facets is not enough to classify the components, new facets can be added by clicking on the add facet button.

The free-text description allows providers to describe the component in natural language. All the specified information is used by the search engine and is viewable by clients after downloading the component information.

Facet	Value
Domain	metrics
Action	parser
Platform	1.3
Safety	good
NeededResource	standalone

The CPPCompleteStream is a parser for C/CPP files, the action of this component is extract metrics from the source file

Figure 46: Specifying classification details for components

The classification details are saved in XML format together with the connectivity description.

A.3.5 Components in the Broker

Once the component is submitted to the broker, the component is stored permanently as an asset of the broker. The application allows providers to add, modify and delete their components exists in the broker. The details of the operations are:

1. Add – add a component to the broker if the component does not exist in the broker. The operation is performed by selecting the Publish command from the right-click pop-up on components listed in the local component list.
2. Modify – replace the component in the broker with the one currently submitted by the provider. The operation is performed by selecting the Publish command from the right-click pop-up on components listed in the local component list.
3. Delete – remove the component in the broker. The operation is performed by selecting the DeleteService command from the right-click pop-up on components listed in the broker component list (Figure 47).

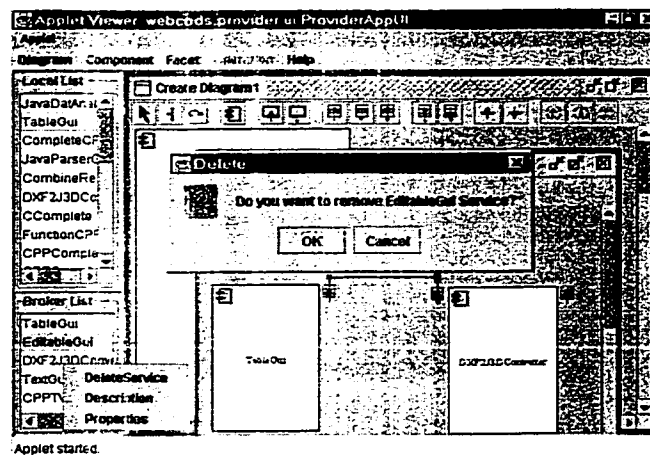


Figure 47: Removing assets from the broker

Appendix B: Description of the Implementation of WebCODS

The package structure of WebCODS is divided according to the existing core elements. There are 8 packages in WebCODS:

1. `Connector` – the templates for the supported connectors
2. `Service` – the operations related to the transfer of components
3. `Parser` – the interpreter for converting textual descriptions to object-based descriptions
4. `Typing` – the representation for components connectivity
5. `Broker` – the implementation for the broker application
6. `Search Engine` – the implementation for the search engine used in the broker
7. `Client` – the implementation for the client application
8. `Provider` – the implementation for the provider application

The following document describes the internals of the WebCODS such that developers can understand the current state of the system, and how to go about fixing or extending it in the future. The overall design is roughly mapped into code, and their implementation details are explained. The packages and classes are discussed.

B.1 Connector

The `connector` package contains templates for the supported connectors. These connectors are used in the composition environment for both the client and provider applications. Each kind of connector has its own template encapsulated in a class. The current implementation of WebCODS supports 4 types of connectors residing in 4 sub-packages.

All connector classes are inherited from an abstract super class called `Connector` residing in the base package. The `Connector` class is used to store the following common information among all subclasses:

1. Store the reference to the object that acts as an input to the connector
2. Save the name of the method that sets up the reference of the connector within the input object
3. Contains the connector parameter

The connector class provides the method interface for `connect`, and `disconnect` methods. The `connect` method is used to connect the input and output objects referred by the connector. The `disconnect` method defines how the connected objects in the connector are separated from each other. The implementation of these methods varies among different types of connectors; therefore, they are declared as abstract and allow the corresponding connectors to provide the actual implementation.

Table 13 lists the package structure of all sub-packages.

Package	Class	Support
<code>webcods.connector</code>	Connector	Base Class for Connectors
<code>webcods.connector.event</code>	Event	Event Connector
<code>webcods.connector.file</code>	File	File Connector
<code>webcods.connector.socket</code>	DatagramSocket	Datagram Socket Connector
	Socket	Socket Connector
<code>webcods.connector.stream</code>	Buffered	Buffer Stream Connector
	NonBuffered	Non-buffered Stream Connector

Table 13: Package structure within the connector package

The supported connection mechanism can be either one-to-one or multicasting. For all the connector classes, they have to:

1. Store the reference to the objects that act as output to the connector
2. Save the name of methods that setup the reference of the connector within all referred output objects

When the connection is one-to-one, the connector needs a reference to the output object and a reference to the input object. By extending the base Connector class, the actual implementation is enhanced with a reference to the output object. When the connection is multicasting, the connector needs references to all the output objects and a reference to the input object. The connector uses a Vector class in the `java.util` package to hold references to output objects.

Connector	Parameter	Connect	Disconnect
Event	None	Passes the reference of the event generating class to all the event listeners	Remove the reference of the event generating class in the event listeners
File	Name and location of the file used to transfer data	Passes the name and location of the file to the input and output objects	Set the name and location of the file to “null” to the input and output objects
Datagram Socket	Socket number	Passes the socket number to the input and output objects	Set the socket number to “-1” to the input and output objects to signal disconnection
Socket	Socket number	Passes the socket number to the input and output objects	Set the socket number to “-1” to the input and output objects to signal disconnection
Buffer Stream	Buffer size	Create a buffered stream of the specified size and passes the reference of the stream to the input and output object	Close the buffered stream and passes a “null” as the reference of the stream to the input and output object
Non-buffered Stream	None	Create a piped stream and passes the reference of the stream to the input and output object	Close the stream and passes a “null” as the reference of the stream to the input and output object

Table 14: Summary for the implementation of connectors

The implementation of `connect`, and `disconnect` methods characterize the properties of the connector. Table 14 shows the implementation details for `connect`, and `disconnect` methods in the supported connectors.

The connectors in WebCODS are templates to connect different components together, the references passing in the `connect`, and `disconnect` methods is based on the reflection feature available in Java.

The steps to pass references between objects using the `java.reflection` package is listed below:

1. For any given input or output object, invoke the `getClass` method to obtain the corresponding `Class` object.
2. Invoke the `getMethod` method on the returned `Class` object with the name of the method that is used to setup the references from the connector. This operation returns the corresponding `Method` object.
3. The `Method` object can be executed using the `invoke` method with the original instance of the object and the arguments.
4. The reference is set in the given input or output object.

The sub-classes of the base class `Connector` also need to provide an implementation of the `isValid` method. The method returns a Boolean value to indicate the validity of the connectivity of the input and output objects referred by the connector. The validity of connectivity is imposed using type-checking and the modes of connection (in, out and bi-directional).

B.2 Service

The `service` package contains the definition for the `CODSEntry` template. The template is used by clients, providers and the broker to transfer components to and from JavaSpaces. The class extends the `net.jini.core.entry.Entry` class from the `Jini` package.

The object referred to inside the template is transferred together with the template to the JavaSpaces; therefore, all referred objects must implement the `java.io.Serializable` to become transferable in the network.

The template posted to the space contains the following information:

1. Recipient of the template, so that messages will not be delivered to unauthorized readers.
2. The message stored in the template is encrypted using the sender's DH private key and receiver's public key.

The information of the recipient is contained in a class called `RecipientInfo` and implements the `java.io.Serializable` interface. The class is responsible for storing the name and company of the user. When perform matching of the recipient in the space, the recipient is matched only if the both the name and company match exactly. The message is in `String` format so that it can be transferred with the template.

The package also contains a classloader called `WebCODSClassloader` to load the classes defined in the `CODSEntry` template. The class is an extension to the `ClassLoader` class available from the `java.system` package.

The `WebCODSClassloader` extends the base class by providing the ability to load classes to the Java Virtual Machine from the component stored in the `CODSEntry` template. Before using the classloader, the provider and client programs decrypt the message stored in the template into a byte array. If the decryption is successful, the byte array contains the executables of the component in a Jar archive. The byte array is passed to the classloader using the `loadClassBytes` method.

The `loadClassBytes` uses the following steps to load the classes into the Java Virtual Machine.

1. The method unjars the Java classes from the jarred byte array
2. Put the obtained classes in the form of byte array
3. For all the obtained classes, uses the `defineClass` method from the parent class to add the binaries to the memory

Figure 48 shows the class diagrams of the service package.

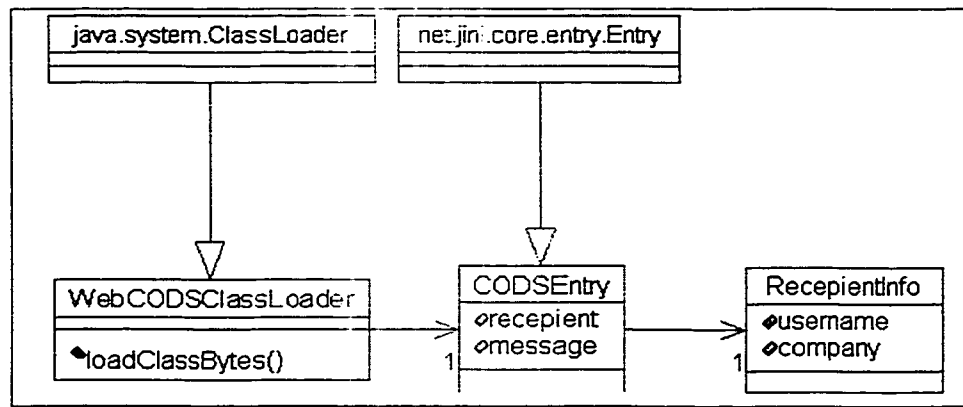


Figure 48: Class Diagram for the service package

B.3 Parser

The textual connectivity descriptions of WebCODS components are analyzed by a textual interpreter generated with JavaCC. The analysis generates the object-based representation of the structure of the component.

The interpreter is used for both primitive and composite components. If the component is a composite component, the interpreter creates the object-based representation for all the referred primitive components. After the basic building blocks are generated, the interpreter is able to add the blocks to the composite component used to encapsulate them. The build composite component is returned as the final product of the analysis.

B.3.1 Creation of Primitive Components

When analyzing the textual description of components, the analyzer first creates a `TypedComponent` object using the name components. The analyzer then performs a check to see if the component is a composite or primitive component. The check is based on the return of the `Class.forName` method. If the method is able to locate the executable for the component, the component is defined to be primitive. Then the name of the component is added to the component table in the `TypedComponent` object.

After the initialization of the `TypedComponent` object, the analyzer proceeds to create the connectivity information of the component. The analyzer reads in the Port description and creates the specified Port Specification. The details of the specification are stored in the `PortSpec` class available in the `webcods.typing` package.

The `PortSpec` specification contains:

1. Name of the port
2. Type of the connection port
3. Direction of the connection
4. The corresponding method name use to setup the parameter for the connection, if the component is a primitive component.

The component may contain more than one port for connections. Each port has its own representation in the `PortSpec` class. The created port specifications are added to the `TypedComponent` object using the `addPort` method.

If the component is primitive, the created `TypedComponent` object is ready to be returned as the final product.

B.3.2 Creation of Composite Components

Composite components contain an implementation section to describe the internal structure of the component.

The implementation section is divided into two subsections.

1. Instance Section – list all the referred components to implement this component
2. Connectivity Section – characterize how the components listed in the instance section are connected together

B.3.2.1 Instance Section

The building of the instance section for `TypedComponent` objects requires a list of components, which is supplied from the composition environment using the analyzer. The list of components contains the connectivity description of all the components currently maintained in the environment. The connectivity description can be in textual format or `TypedComponent` representation.

The implementation section allows multiple instances of the same component to build up the component. The structure used to specify components contains two details to declare an instance of components:

1. An instance name
2. The name of the component.

Given the name of the component, the list of components is consulted to obtain the description of the component. If the list contains the `TypedComponent` object for the component, the instances of components are added to the `TypedComponent` object using the `addCompositeComponent` method. The `addCompositeComponent` method takes in the `TypedComponent` and the instance name as parameters. If the textual description for the component is available, the description is analyzed to obtain the object-based representation. The description is then added to the `TypedComponent` representation.

The instance parameters for the component can also be added to the `TypedComponent` representation of the component using the `setInstanceParameter` method.

B.3.2.2 Connectivity Section

After the instantiation of the required `TypedComponent` object referred to this composite component, components can connect together according to the `CONNECT` constructs.

In each `CONNECT` construct, the following information is defined:

1. The name of input ports to the connector
2. The names of output ports to the connector
3. The type of connection
4. The specific connector used in the connection
5. The parameter of the connector if required
6. The instance parameter for the connector if specified

The name of the ports specified in the connect constructs is composed of the instance name and the name of the port in the component. The instance name for the component is based on the information created in the instance section. Using the `getCompositeport` method available from the `TypedComponent` object, the corresponding `PortSpec` representation of the port can be resolved.

When the analyzer obtains the specified connector from the description, the required connector is created. The obtained `PortSpec` representations for involved ports are added to the created connector. The adding of the `PortSpec` object uses the `setInPort` and `setOutPort` methods according to specification of the construct. The established connector is then added to the description using the `addConnector` method.

The instance parameters for the component can also add to the connector using the `addConnectorParameter` method.

After setting up the instance and connectivity section, the `TypedComponent` object can be used to represent the composite component defined in the textual description.

Table 15 summarizes the use of different sections in the textual description to create `TypedComponent` objects for primitive and composite components.

Section	Primitive Component	Composite Component
Component Name	Corresponding to Java executables	Any name
Ports	Same as composite components. The specification contains the name of the method to be used by the connector	Specifies the name, operation direction and type of the port.
Instance	N/A	Create an instance section for the referred components. All the referred components are resolved into <code>TypedComponent</code> representation
Connectivity	N/A	Connect the components listed in the instance section. Instance of <code>PortSpec</code> objects are added to the <code>Connector</code> object to be connect in the composition environment.

Table 15: Summary for steps required to create `TypedComponent` objects.

B.4 Typing

The `TypedComponent` class in the package is used to represent the description of a component into a set of Java objects. The representation can be used for both types of components. Descriptions are stored in a wrapper object called “Typed Component”. This wrapper object is used to store (1) the descriptions of all referred subcomponents (2) the information of the interfaces, (3) the connectivity information within the abstract component.

The `TypedComponent` class contains the following lists to store the details used to represent the component.

1. Port list – the list of ports available for connection
2. Instance list – the list of required instances of components
3. Connector list – the list of connectors used to connect internal components
4. Component Parameter List – the list of parameters to be used by internal components
5. Connector Parameter List – the list of parameters to be used by connectors
6. Object List – the list to store the references to instances of Java executables referred by internal components

Figure 49 shows the class diagrams of the `typing` package.

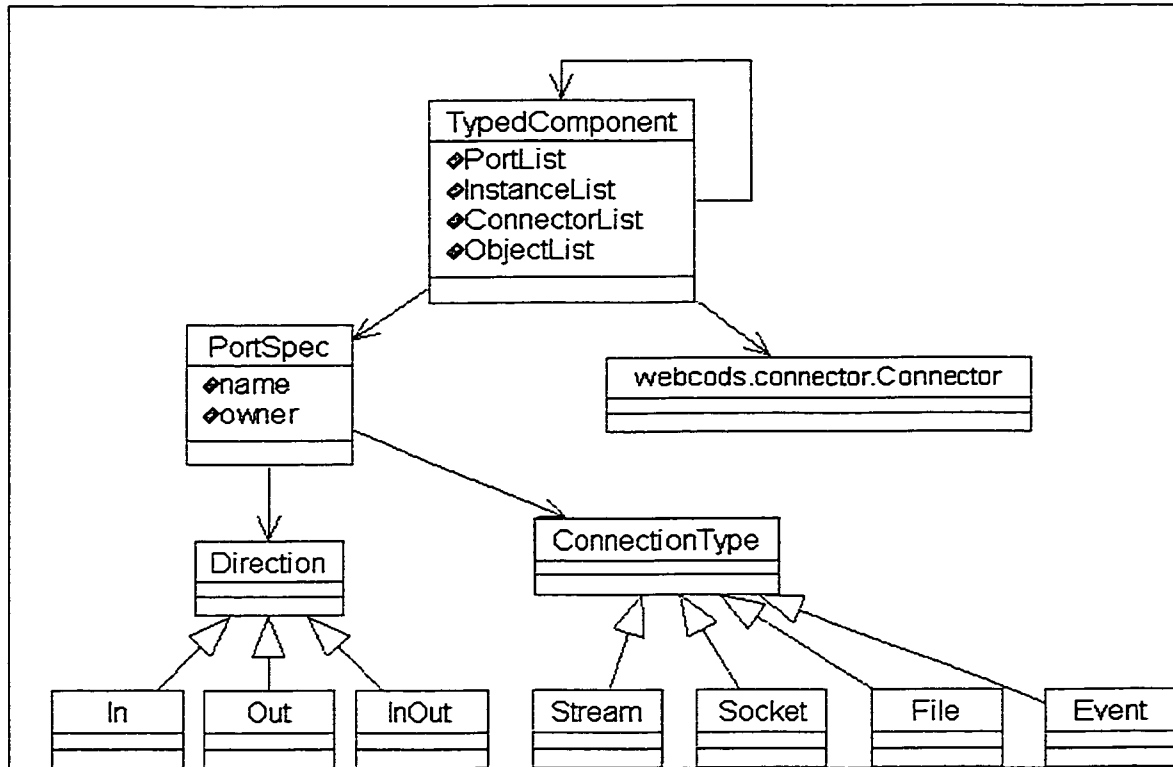


Figure 49: Class Diagram for the `typing` package

The usage of other supporting classes in the `typing` package is listed in Table 16.

Class Name	Usage
<code>PortSpec</code>	Hold the name of the connection port and connection details: mode of operations (directions) and type of connections.
<code>Direction</code>	Base class for defining the mode of operation
<code>In</code>	Subclass of <code>Direction</code> – indicates the mode of operation is IN
<code>Out</code>	Subclass of <code>Direction</code> – indicates the mode of operation is OUT
<code>InOut</code>	Subclass of <code>Direction</code> – indicates the mode of operation is IN/OUT
<code>ConnectionType</code>	Base class for defining the type of connection
<code>Event</code>	Subclass of <code>ConnectionType</code> – specifies the connection is of type Event
<code>File</code>	Subclass of <code>ConnectionType</code> – specifies the connection is of type File
<code>Stream</code>	Subclass of <code>ConnectionType</code> – specifies the connection is of type Stream
<code>Socket</code>	Subclass of <code>ConnectionType</code> – specifies the connection is of type Socket

Table 16: Usage of supporting classes in the `typing` package

The `TypedComponent` class and all other referred classes implement the `java.io.Serializable` interface. Instances of `TypedComponent` class are transferred in the network with the required executables of components.

When `TypedComponent` objects arrive in the composition environment, the objects can be resolved into an executable instance of the represented components. The resolving of `TypedComponent` objects achieves the following purposes:

1. Create running instances of executables referred in the primitive components
2. Obtain references to these running instances and put them into the object list
3. Connect the components together using instances of executables

After establishing the object list, the list can be obtained from the `TypedComponent` object. Since all the main classes of components in WebCODS are extending the `java.lang.Thread` class, the components in the list can be executed by calling the `run` method.

The resolving to executables in the `TypedComponent` is based on the `resolve` method. The method relies on a classloader for the class to locate the required executables of the components.

In the provider environment, the WebCODS components are installed locally in the machine. The Java classes can be obtained from the classloader supplied from the Java Virtual Machine. However, in the client environment, the executables are stored in a byte array. The contained Java classes in the array require the using of the `WebCODSClassloader` to read the byte array and load the classes and other class from the system. Therefore, the method requires the user of the method to supply the appropriate classloader to load classes.

The resolve mechanism for composite and primitive component is different. The first step for the resolve is to check the type of components using the `isPrimitive` method. The component represented by the object is primitive, if the implementation section does not exist in the description. Therefore, the querying of the size of the instance list is sufficient to derive the type of the component.

B.4.1 Resolving of Primitive Components

The resolving of primitive components proceeds with the following steps:

1. Use the supplied classloader to obtain the Java class
2. Create a new instance of the class
3. Obtain the specified parameters using the `getParameter` method
4. Invoke the `createInstance` method available in the `execute` class of the component to obtain an instance with the specified parameters
5. Setting up the referred `PortSpec` objects to refer to the corresponding executed instance.

After the creation of a new instance of the component, a reference to the component to the object table is added to the object table of the `TypedComponent`. The component is waiting to be executed by the composition environment

B.4.2 Resolving of Composite Components

The resolving of composite components proceeds with two stages. The first stage is the resolving of the components listed in the instance list. The second stage establishes connections stored in the connector list.

In the first stage, each component in the instance list goes through the following steps:

1. Pass parameters used in instantiation to the `TypedComponent` object
2. Resolve the component using the supplied classloader
3. Obtain the references to the instances of the executables contained in the resolved component
4. Add the references to the object table in the `TypedComponent` object

After resolving all referred primitive and composite component, the object table holds references of all the executables required by the component. However, the component is not connected with using any connectors in the connector list.

The second stage is to use the `Connector` object in the connector list to connect components together. Each connection in the list requires the following steps:

1. Pass specified parameters to the `Connector` object
2. Verify the validity of the connection
3. Connect the objects together using the `connect` method available in the `Connector` class

The final step for stage two is setting up the `PortSpec` objects contained in the component. The first step to setup the object is to locate the `PortSpec` object referred in its subcomponents. From the located `PortSpec` object, a reference to the corresponding executable of the component can be obtained. The reference is passed to the `PortSpec` object that needs to be initialized -- the `setOwner` method is invoked with the reference to the executable as the argument.

B.4.3 Textual Description

After the resolving of `TypedComponent` objects component, the component is able to query itself for a textual description. The textual description is generated by invoking the `getDescription` method.

The generation of the description relies on the ability of the `Connector` and `PortSpec` class to generate a textual description to describe its internals. The steps used to create the textual description is listed below:

1. Obtain the name of the component
2. Prepare the first line in the description
3. For all the `PortSpec` objects contained in the port list, invoke the `getDescription` method to obtain a textual representation of the port.

If the type of component is primitive, the preparation of the textual description is finished. In the case of composite components, the textual description also contains an instance and a connectivity section. The preparation of the description continues:

1. The instance list of components is visited to find out names of the instance and the names of the component.
2. If parameters for the instance are specified, the list of parameters is attached at the end of the INSTANCE construct.
3. The connection list of component is consulted to produce the textual description of all the connectors.
4. If a parameter for the connector is specified, the parameters is attached at the end of the CONNECT construct.

B.5 Broker

The `broker` package contains the implementation for the component broker in WebCODS. The broker is a Java application running as a server for the application. It holds the following responsibilities:

1. Authenticate clients and providers to login to the broker
2. Connect clients and providers to the broker
3. Receive components from providers
4. Dispatch components to clients
5. Maintain the list of components received from providers
6. Perform backup services of the component list

The `broker` package is further divided into the following sub-packages:

1. `dataaccess` – the related operations to access the database
2. `clientsession` – the session for the client in the server
3. `providersession` – the session for the provider in the server
4. `deliveryservice` – delivery service to transfer components between clients and providers
5. `security` – the encryption and decryption services required in the delivery service
6. `servicecenter` – the service centre maintains the list of components in the server

B.5.1 Data Access

The UML class diagram for the package is shown in Figure 50. The package consists of three classes:

1. `AccessManagementData` – the class extends the `AccessData` class and provides methods to verify the usernames and passwords obtained from the database.
2. `AccessData` – the class contains the core method used to access the database through the `DataStorage` class
3. `DataStorage` – the proxy class to the database

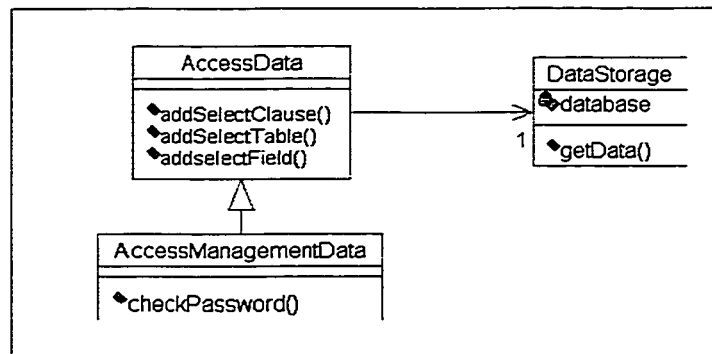


Figure 50: UML class diagram for the `accessdata` package

The verification of usernames and passwords is based on the `checkPassword` method available in the `AccessManagementData` class. The method returns a Boolean value to indicate the validity of the supplied username and password. The parameters passed to the method includes a username, a company name and the supplied MD5 digest of the message. The MD5 digest is created using the username, the company name and the password of the user. In this way, the password is never dispatched to the network.

After receiving the supplied information, the method builds up a customized SQL clause to select the password of the user from the database. The SQL is built using the supplied operations in the `AccessData` class, such as, `addSelectClause`, `addSelectTable`, and `addSelectField`. The usernames and passwords are stored in a table called `Users` in the database. When setting up the `AccessData` class to retrieve the `User` table, the `Password` field of the specified user can be obtained.

The verification of the supplied user information starts after retrieving the corresponding password from the database. The program recalculates the MD5 message digest using the received username and company name, together with the password obtained from the database. The calculated digest is compared against the received digest. If both digests have an exact match, the login information is supplied from an authenticated party. The method returns a true value to indicate the success of the validation.

The creation of the `AccessData` class initializes the connection to the database using the `DataStorage` class. The class connects to the database installed in the system using the JDBC Bridge. The class also provides an accessing method to the database. The accessing method is called `getData`. The method takes in the following parameters:

1. A vector of interesting field
2. A vector of interesting table
3. A vector of selecting clause

The method creates an SQL statement in the form of a `Statement` class and passes it to the database. The queried result is in the form of the `ResultSet` class. The `Statement` and `ResultSet` classes are defined in `java.sql` package.

B.5.2 Sessions in Server

The broker application is implemented as an RMI server. The authentication of clients and providers are based on RMI.

When the server is initialized, it registers itself to a RMI registry as “Broker”. In the HTML page for the clients and providers, they specify a name and the location to locate the broker. They can then connect to the broker using RMI and perform the authentication.

After being authenticated by the broker, the broker creates new sessions for the connected party. If the connected party is a client, the established session is a client session. The interface of the session is defined in `ClientSessionInterface`. If the connected party is a provider, the created session is a provider session and has an interface defined in `ProviderSessionInterface`. The detailed descriptions of the client session and provider session are summarized in Table 17.

Session	Method	Usage
Client	<code>requestService</code>	Submit a request for components to the broker
	<code>getComponentInfo</code>	Request description of a component in the broker
	<code>Query</code>	Query the broker with the supplied requirements
Provider	<code>addService</code>	Add a new service to the broker
	<code>modifyService</code>	Modify the service existing in the broker
	<code>removeService</code>	Remove the existing service in the broker
	<code>listService</code>	List the submitted components of the broker
Common	<code>getFacets</code>	Obtain the facets defined in the search engine

Table 17: Summary of the sessions created in the broker

Figure 51 shows the UML description for the architecture of the broker.

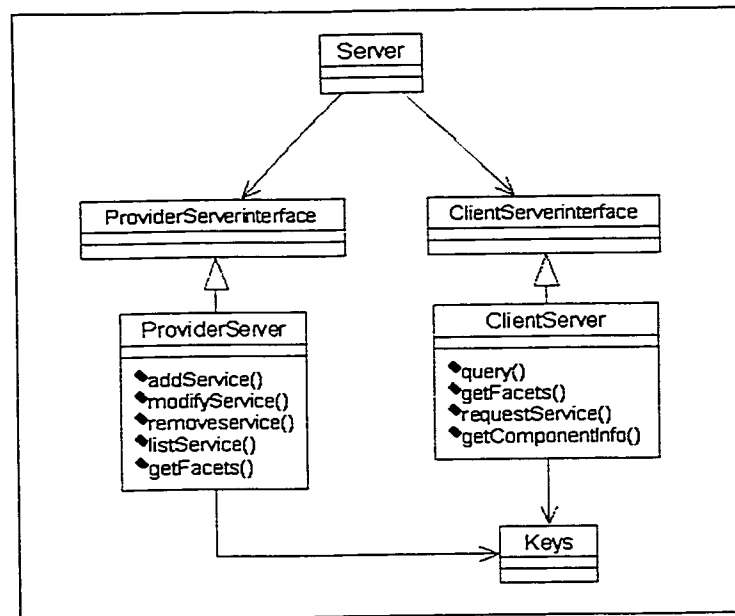


Figure 51: UML class diagram for the broker package

Given the users' information, the corresponding public key can be identified from the system keystore file. Using the same information, the system is also able to find the DH public key of the party in the secret keystore file. If the DH public for the party does not exist, the session will request the party to generate a new key set. After the key is generated, the generated public key is transferred to the broker and is stored in the secret keystore file.

The generated session holds the retrieved public keys of the connected party:

- Public key
- DH Public key

The keys are required by the security center when performing encryption, decryption and verification of digital signature.

B.5.3 Delivery Service

The `DeliveryService` class is responsible for obtaining a reference to the `JavaSpaces` that is used for delivering components. The initialization of the class uses the `LookupLocator` class to find a `Register` for the Jini server. The lookup locator reads in the location of the Jini server from the INI file. The `Register` is then used to identify an instance of the `JavaSpaces` services with the specified name also from the INI file. The reference to an instance of the `JavaSpaces` is stored as an attribute in the class.

When clients request for components, the client session will consult the service center to deliver the component. The service center obtains the component from the component list. After obtained the component, the component is transferred to the security center for encryption and digital signing. The component then sends to the delivery service for transferring to the client.

The `DeliveryService` class uses the `delivery` method to send the component to the space. The method requires information of the receiver and the encrypted component as arguments. The delivery process has the following steps:

1. Prepare an instance of the `RecipientInfo` that contains the receiver's information
2. Prepare an instance of the `CODSEntry` that stores the `RecipientInfo` and the encrypted component
3. Write the instance of the `CODSEntry` to the `JavaSpaces`
4. Return an `Acknowledgement` to the client

B.5.4 Security Center

The security center performs encryption, decryption and digital signing of components. The operations are dependants on the private and public keys of the broker and the public keys obtained from the corresponding established session.

The main methods available in the `Security` class are:

1. `receiveData` – decrypt the data using the keys
2. `sendData` – encrypt the data using the keys
3. `startDHSession` – calculate the DH secret key for the session

When data is received from client sessions or provider session, the `receiveData` in the `Security` class is consulted. The method takes in the encrypted message, and the user name of the session as parameters. The decryption of the message is based on the following steps:

1. Verification of the signature contained in the message – use the `verify` method in `SignedObject` to compare the signature used to sign the message and the signature created with the public key stored in the session.
2. Generation the key for decryption – the message is encrypted with a key generated using the DH private key of the encrypter and the DH public key of the decrypter. Based on the properties of DH key agreement, the same key can be generated from using the DH private key of the decrypter and the DH public key of the encrypter. Therefore, the decryption key can be generated using the DH private key of the broker and the public DH key stored in the session using the `startDHSession` method.
3. Decryption – when the agreed DH secret key is generated, it can be used to decrypt the message.

The encryption is the reverse of the decryption process described above. The sending out of a component uses the `sendData` method. The method takes in the message, and the user name of the session as parameters. The encryption of the message is based on the following steps:

1. Generate the key for encryption – the decryption key can be generated using the DH private key of the broker and the public DH key stored in the session.
2. Encrypt – when the agreed DH secret key is generated, it can be used to encrypt the message.
3. Sign – using the public key and create the `SignedObject` for the message

The `startDHSession` is used to create the secret DH agreement key. The procedure to create the key is listed below:

1. Create an instance of `KeyFactory` class using DH key agreement
2. Initialize the `KeyFactory` instance with the DH private of the broker using the `init` method.
3. Add the public key of the session to the `KeyFactory` instance with the `doPhase` method.
4. Generate the secret key using the `generateSecret` method
5. Create an instance of the `SecretKeyFactory` using “DESede” algorithm
6. Initialize an instance of the `DESedeKeySpec` with the previously generated secret key
7. Regenerate the secret key using the `generateSecret` method available in the `DESedeKeySpec` class

B.5.5 Service Center

The service center contains a list of the components received from providers. The service center is the point of interaction for the established session in the broker. The service center interface provides the following functions:

1. `addService` – add a new component to the component list
2. `modifyService` – modify an existing component in the component list
3. `removeService` – remove an existing component in the component list
4. `query` – search components using the search engine

The component being added to the component list contains:

1. Executables
2. Object-based representation of the component
3. Classification details in XML format

When components are added to or modified in the component list, the XML description is added to the search engine. The service center delegates the query operations to the search engine that contains the classification details.

The service center is also responsible for the backup of the component list. If the broker crashes, the component list can be reloaded and continue serving components to clients. The backup of the server is performed in the `ServiceCenterBackup` class.

The backup of the broker proceeds with the following steps for each component in the list:

1. Save the executables and typing descriptions
2. Store the XML descriptions

The information of the component is written to an `ObjectOutputStream` and saves to a file. The backup mechanism is implemented as a `Thread`, and it is activated every 10 minutes.

The rebuilding of the broker simulates the process of submitting components from providers; each component in the saved list is added to the reinitialized broker sequentially. In this way, the component list of the broker is restored to the last saved execution state and the status of the search engine can also be rebuilt.

The UML description of the `servicecenter` package is shown in Figure 52.

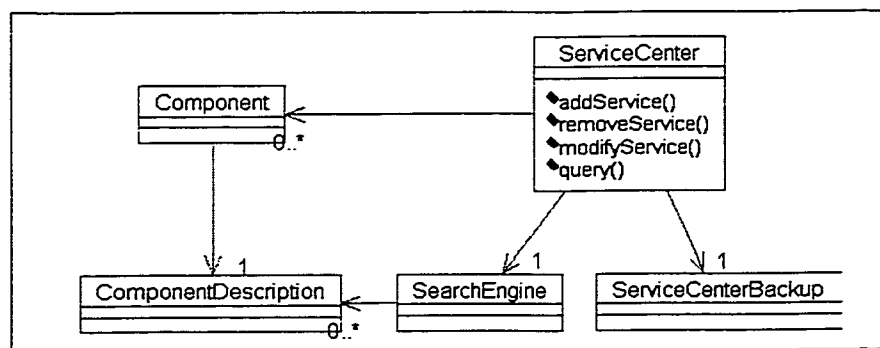


Figure 52: The UML description of the `servicecenter` package

B.6 Search Engine

The core functionality of the engine is listed in the `Engine` interface. This interface requires implementing classes to provide the following methods:

1. `search` – the finding of components in the engine
2. `addDocument` – add new description to the engine
3. `modifyDocument` – modify the existing description in the engine
4. `removeDocument` – remove the existing description in the engine
5. `getNumComp` – return the number of descriptions available in the engine

When the search engine is created, it reads in a few files to establish the internal structure.

1. Facets file – the file contains the list of facets used in the classification of components
2. Thesaurus files – each of the facets in the Facets File may be accompanied by a thesaurus file. The thesaurus file contains synonyms and their related weights.

The search engine supports searching in the following ways:

1. List the components belongs to a provider using the `searchProvider` method
2. Match of attributes in facets using the `searchFacets` method
3. Search of the query in the free-text description using the `searchDescription` method

The searching mechanism in the `searchFacets` and `searchDescription` methods are dependants on the thesaurus created during the initialization of the search engine. The queries supplied to these methods are expanded using the search method available in the `Thesaurus` class. The expanded result is listed in `ResultPath` class. Because the class extends `java.util.Iterator`, the `next` method can be used to traverse the returned result. Each entry in the result contains the related synonym and its proximity related weight.

The thesaurus is represented as a directed graph with weighted links. Each node in the graph is represented as a `GraphElement`. This class contains the represented synonyms and their proximity relationship. The `Thesaurus` class contains a `java.util.Hashtable` to store the `GraphElement` objects. This class has a recursive `performSearch` method that can transverse the graph and identifies all related nodes for the supplied query. The use of the thesaurus allows identification of one or more related components with ranked closeness according to the input query.

When searching components using the thesaurus, there is a possibility that all terms in the thesaurus are related. Therefore a threshold value is required in the `performSearch` method to eliminate the not-so related synonyms.

The UML description of the `searchengine` package is shown in Figure 53.

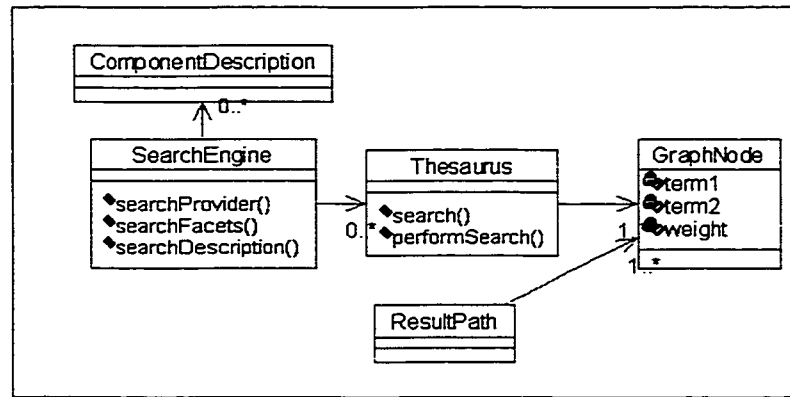


Figure 53: The UML description of the `searchengine` package

The `searchengine` package also contains an application for the system administrator to build and test the thesaurus. The application contains the following classes:

1. `ThesaurusViewer` – the main class of the application. The class is able to read in a thesaurus file and allows modification to the thesaurus. After testing of the thesaurus, the modification can be written to a thesaurus file and used by the application.
2. `VisualizePanel` – the class creates an instance of the `Thesaurus` class for testing. The user of the application can perform queries on the created thesaurus and see the result.
3. `QueryMenu` – the class used to create a query menu to search the thesaurus. The query menu allows the user to supply a query string and a threshold to be used in the search.
4. `ResultViewer` – the class used to view the result of the querying of the thesaurus. The result lists the related synonyms and their weighted proximity relationship.

B.7 Client Application

The client application is implemented as a Java Applet. The main execution class, `webcods.client.ui.ClientAppUI` of the application extends the `java.swing.JApplet` class. The package contains four main components:

1. Initialization of the client application – the `webcods.client` package
2. Creation of the user interface of the application – the `webcods.client.ui` package
3. Composition environment for components – the `webcods.client.ui.composediagram` package
4. Searching and downloading components – the `webcods.client.ui` package

B.7.1 Initialization

Initialization of the client application is required to establish connection with the broker. The application opens the keystore file to locate the public and private key set of the client, and the public key of the broker.

The connection process starts with obtaining a reference to the server using the name of the broker and the location of the registry server. After obtained a RMI connection with the server, the client supplies the username and password to establish a client session with the broker.

The username and password is used to generate a MD5 code. The username and the created MD5 code are encapsulated as a `SignedObject` object. The object is digitally signed using the private key of the client. The prepared message is sent to the broker for verification.

B.7.2 UI Creation

The application UI is implemented using Swing components. It is composed of the following components:

1. `DownloadedComponentList` – based on `java.swing.JList`. The list contains components downloaded from the broker.
2. `ToDoPane` – based on `java.swing.JDesktopPane`. The panel contains sub-frames to perform component composition.

The `DownloadedComponentList` stores the list of components downloaded from the broker. The list displays the name of components, but the model behinds the list stores all the details of the downloaded components:

1. Executables
2. `TypedComponent` representation
3. XML description

The list also supports operations to visualize properties for the contained components:

1. View connectivity description of components – the `ViewPanel` class. The class uses a `JTextPanel` to view the textual description for the component.
2. View classification description of components – the `ViewPropertiesPanel` class. The class uses a `JPanel` to view the facets attributes and free-text description for the component.

The `ToDoPane` provides frames to compose components – `ComposeDiagramFrame`. The frame is supported by a `ComposeDiagramModel` in the `webcods.client.ui` package.

When components are dropped from the `DownloadedComponentList` to the `ComposeDiagramFrame`, the component is loaded into the composition environment with the following steps:

1. Create a new `ComposeDiagramFrame` in the `ToDoPane`
2. Save descriptions of applications created in the `ToDoPane`
3. Load saved descriptions into the `DownloadedComponentList`
4. Exit the application
5. Display the about box of the application

The main UI class containing the `DownloadedComponentList` and `ToDoPane` is called `NavigatorPane`. The panel creates the main menu for the application. The `Action` class is the implementation for the menu.

The menu contains the following actions:

6. Creation of new `ComposeDiagramFrame` in the `ToDoPane`
7. Saving descriptions of applications created in the `ToDoPane`
8. Loading of saved descriptions into the `DownloadedComponentList`
9. Existing the application
10. Displaying of the about box of the application

B.7.3 Compose Diagram

The `composediagram` package contains the implementation for the composition environment. The implementation is divided into two areas:

1. Analysis – the composition model of the environment
2. Drawing – the displaying of the components in the environment

When components are put on the `ComposeDiagramFrame`, the component is represented as a `ComponentNode`. The type description of the component is analyzed by the `ComponentNode` and displays the component on the `ComposeDiagram`. The drawn component is added to the `ComposeDiagramModel`.

The `ComponentNode` contains a `TypedComponent` of represented component. The representation is resolved to contain the executable instances of referred components. The node has access to the internal structure of the component therefore the connectivity information of the component can be realized. The component is displayed as a box with ports surrounding it to indicate its connectivity. The `PortsSpec` objects in the description are represented by class of type `ConnectPort`. Each kind of port has its own implementation of this class. For example, the Stream port has a `StreamPort` class to define the behavior of the port. The behavior is restricted by the permissible type of connector links to the port and the directions of operation.

The model of connection between components is represented by `ConnectEdge`. The class is responsible for:

1. Creating the correct representation of the connecting edge figure
2. Checking the permissibility of the connection between the ports

The `ComposeDiagramModel` is the container of the composed application. It stores references to the contained `ComponentNode` and `ConnectEdge` objects. When connections are request by the composer in the composition environment, the model is consulted to verify the permissibility of connection. The verification process is delegated to the corresponding `ConnectPort` and `ConnectEdge` objects for accurate analysis.

Because the `ComposeDiagramModel` contains the model of the application, textual description for the composed application can be generated from the model. The created components in the panel must be composite components.

The creation of description is based on the following steps:

1. Obtain the name of the component
2. Prepare the first line in the description
3. From the `ComposeDiagramModel`, obtain all unconnected `ConnectPort` objects and then invoke the `getDescription` method to obtain a textual representation of the port.
4. Visit the contained `ComponentNode` in the model and generate unique instance names for the components.
5. If parameters for the instance are specified, attach the list of parameters is to the end of the `INSTANCE` construct.
6. Consult the list of `ConnectEdge` objects to obtain the corresponding connectors for connections. Create the textual description using the `getDescription` method of all the connectors.
7. If a parameter for the connector is specified, attach the parameters to the end of the `CONNECT` construct.

Component drawing is accomplished using drawing classes provided by the GEF package. Each drawing element in the diagram extends the `EdgeFig` class from the GEF.

Table 18 lists the drawing classes in the composition diagram and their usage.

Class	Usage
<code>DatagramSocketEdgeFig</code>	Contains the datagram socket connector and creates the menu for input socket number
<code>EventEdgeFig</code>	Contains the Event connector
<code>FileEdgeFig</code>	Contains the file connector and creates the menu for input file name
<code>StreamEdgeFig</code>	Contains a menu for composers to select the preferred pipe style. The class also stores the selected connector and its parameters.
<code>SocketEdgeFig</code>	Contains the socket connector and creates the menu for input socket number
<code>FigIcon</code>	Drawing class for icons to represent ports surrounding the component

Table 18: Supporting classes for drawing in the composition diagram

The menus to set up parameters are stored in the `ui` package under `composediagram`. Each connector has a specific implementation of the menu, even for those do not require a menus to input parameter. The menu is used for

connecting and disconnecting the connection between components, and the generation of textual descriptions for the connectors.

B.7.4 Searching and Downloading Components from the Broker

The application also has the capabilities to find components and download components from the broker.

B.7.4.1 Searching

The functionality to search for components in the broker is encoded in the `MakeQuery` class. The class uses a `JDialog` for clients to input the matching criteria. The criteria includes searching for:

1. Provider's Name
2. Facet's Name
3. Free-text description

The facets list is obtained dynamically from the broker by invoking the `getFacets` methods available from the `ClientSessionInterface`. After specifying the query, the information is gathered into a text string and submits to the broker.

The `ClientSessionInterface` provides a search method to query the search engine. The return of the method is the XML description of all the matched components. The XML description is viewed in a `DisplayQuery` class. The viewing panel is an extension of the `ViewEditorPanel` that is configured to read in HTML. The configured panel is able to view XML descriptions in a tabulated format.

The listing of components is a similar process to searching. However, the query of the search is not specified, it is able to match all the listed components in the broker.

B.7.4.2 Downloading

The components listed in the `ViewEditorPanel` after listing or searching can be downloaded to the composition environment. This process is initiated by issuing a `requestService` command to the `ClientSessionInterface`.

The `requestService` method takes in the name of the component as parameter. The return of the method is an `Acknowledge` from the broker. The class contains two required objects for the client to obtain the component:

1. `JavaSpaces` – a proxy to the `JavaSpaces` that the broker uses
2. `CODSEntry` – a prepared template for the client to match the component from the space.

After obtaining the `Acknowledge`, the client can issue the `take` command to obtain the requested component. The component is stored within the `CODSEntry` and encrypted with the agreed DH key between the client and the broker.

The package contains a `Security` class that is responsible for the management of keys in the application. The main methods in the class are `decrypting` and `startDHSession`. The `decrypting` method is used to decrypt the encrypted message in the `CODSEntry` obtained from the space using an agreed key generated from the `startDHSession` method.

The decrypted message contains the executables of the component and the connectivity description. The executables are loaded to the environment using the `WebCODSClassloader`. The connectivity description of the component is added to

the `DownloadedComponentList` and displayed in the panel. When the component is put in the composition environment, a cloned description for the component is returned.

B.8 Provider Application

The provider application is implemented as a Java Applet. The main execution class, `webcods.provider.ui.ProviderAppUI` of the application extends the `java.swing.JApplet` class. The package contains four main components:

1. `webcods.provider` - Initialization of the provider application
2. `webcods.provider.ui` - Creation of the user interface of the application
3. `webcods.provider.ui.creatediagram` - Creation environment for primitive components
4. `webcods.provider.ui.composediagram` - Composition environment for components

B.8.1 Initialization

The process is similar to the initialization of the client application. The connection process is started with obtain a reference to the server using the name of the broker and the location of the registry server. The provider supplied the company name and password to establish a provider session in the broker.

The generated username and the created MD5 code are encapsulated in as a signed object using the private key of the provider. The prepared message is sent to the broker for verification.

B.8.2 UI Creation

The interface of the application is implemented using Swing components. It is composed of the following components:

1. `LocalComponentList` – based on `java.swing.JList`. The list contains components downloaded from the broker.
2. `ServerComponentList` – based on `java.swing.JList`. The list contains components loaded to the broker.
3. `ToDoPane` – based on `java.swing.JDesktopPane`. The panel contains sub-frames to perform component creation and composition.

The `LocalComponentList` stores the list of components previously created in or loaded into the application. The list displays the name of components, but the model behind the list stores all the details of the local components.

1. Name of the component
2. Location of the package of the component (for primitive component)
3. Location of the jar file that contains the executable (for primitive component)
4. Location of the connectivity description for the component
5. Location of the classification description

The information is retrieved when the component is dropped to the composition environment or publishes to the broker. When the application exits, the details of components stored in the `LocalComponentList` are saved into a XML formatted file. The file is saved to the current user's home directory. The initialization of the `LocalComponentList` loads the saved component information and displays them in the list.

The list also supports operations for the contained components:

1. View connectivity description of components – the `ViewPanel` class. The class uses a `JTextPanel` to view the textual description for the component.
2. View classification description of components – the `ViewPropertiesPanel` class. The class uses a `JPanel` to view the facets attributes and free-text description for the component. The panel also supports the modification of the classification details. The updated details are saved in the description file.
3. Remove the component from the list
4. Publish the component to the broker

The `BrokerComponentList` lists the components previously submitted by the provider. The process is the same as querying the broker using the provider's name. The queried result is saved in the list. The list stores the following information:

1. Name of the component
2. Classification details of the component – in XML format

The list supports the remove and view operations for the contained components:

1. Remove – delete the component from the broker. The remove includes the removal of the executables and the corresponding descriptions.
2. View connectivity description – obtain the description based on the information obtained from the `LocalComponentList`
3. View classification description – view the classification description of the component returned from the querying of the search engine.

The `ToDoPane` provides frames to create and compose components – `CreateDiagramFrame` and `ComposeDiagramFrame`. `CreateDiagramFrame` creates new primitive components. The `ComposeDiagramFrame` composes new components from the `LocalComponentList`. After new components are created, they are added to the list, and are ready to be published to the broker.

When components are dropped from the `LocalComponentList` to the composition environment, the executables are loaded to the system memory using the system classloader. The component is drawn on the corresponding frame for manipulation.

After new components are created in the `ToDoPane`, the classification details of components are input using the `ViewPropPanel`. The panel obtains the facets from the broker using the `getFacets` methods from the `ProviderSessionInterface`. Since the list is dynamically added to the application, the modification of the classification scheme does not need to reprogram the panel.

The main UI class containing the `LocalComponentList`, `ServerComponentList` and `ToDoPane` is called `NavigatorPane`. The panel creates the main menu for the application. The `Action` class is the implementation for the menu.

The menu contains the following actions:

1. Create new `CreateDiagramFrame` in the `ToDoPane`
2. Create new `ComposeDiagramFrame` in the `ToDoPane`

3. Load saved descriptions into the `LocalComponentList`
4. Exit the application
5. Display the about box of the application

B.8.3 Creation Diagram

The `creatediagram` package contains the implementation for the creation of new and modification of existing primitive components.

When components are put on the `CreateDiagramFrame`, the component is represented as a `ComponentNode`. The type description of the component is generated by the node and displays the component on the `CreateDiagram`. The drawn component is added to the `CreateDiagramModel`.

The node has access to the `TypedComponent` representation of the component and displays the ports using the corresponding icons. When the node is representing a new component, the `TypedComponent` refers to a `NULL` object. Ports can be added or removed from the `ComponentNode`. The ports drawn in the diagram are represented as `PortImage`.

The available ports for adding to components are listed in the menu of the diagram. The menu is created using the following classes:

1. `CmdMenuIn` – loads the icons for ports supporting in direction.
2. `CmdMenuOut` – loads the icons for ports supporting out direction.
3. `CmdMenuInOut` – loads the icons for ports supporting in and out direction.

Table 12 summarizes the icons used for ports in the creation and composition environment.













Port	Direction		
	<i>In</i>	<i>Out</i>	<i>In/Out</i>
Stream			N/A
File			N/A
Event			N/A
Socket			
Datagram Socket			

Table 19: Icons for ports in WebCODS

The `PortMenu` class in the `menu` package is supplied for composers to input information, when adding or modifying of ports in the diagram. The dialog requires composers to provide the following information:

1. Name of the port
2. Select a method exists in the main class to set up the reference for the connection

The `CreateDiagramModel` is the container of the created primitive component. It stores references to the contained `ComponentNode` object. The model also stores references to `PortImage` objects in the diagram. The model is capable of generating the textual description for the created component using information supplied by composers and contained information in the model.

The creation of description is based on the following steps:

1. Obtain the name of the component
2. Prepare the first line in the description
3. From the `CreateDiagramModel` obtains all `PortImage` objects, and then invokes the `getDescription` method to obtain a textual representation of the port.

After the textual description is prepared, the component creator needs to specify the following information using the `ComponentMenu` dialog. The dialog allows the provider to specify:

1. The main class to execute the component
2. The path to the package of the component
3. The path to the jar file that contains the executables

When the component is saved to `LocalComponentList`, the textual connectivity description is saved in the same directory with the main execution class. The other details specified in the `ComponentMenu` dialog are stored in the list and saved in the XML file in the user directory.

B.8.4 Compose Diagram

The `composediagram` package contains the implementation for the composition environment.

The implementation is the same as the `composediagram` in the client package.

B.8.5 Submitting Components to the Broker

The components listed in the `LocalComponentList` can be submitted to the broker environment. The process is initialized by issuing a publish command in the list. The command is passed to the `ProviderSessionInterface` using the `addService` method.

The publish command picks up the selected component in the `LocalComponentList`. The command is transferred to the `NavigatorPane`. The panel obtains a reference to the `ServerComponentList` and invokes the `addElement` method. The method takes in the name of the component as the input parameter. The request is then submitted to the `addService` method in `ProviderAppUI`. Given the name of the component, the XML description and the path to the component can be realized.

With all required details for the component, the `addService` method in `ProviderApp` is invoked. The method prepares the component in a secure and transferable format for the network.

The preparation of the details required for the component is listed below:

1. Read in the textual description of the component
2. Analyze the textual description of the component using the `TypeParser` in the `parser` package
3. Resolve the `TypedComponent` object using the system class loader
4. Obtain the list of referred primitive components saved in the object table
5. Add the required executables into a new jarred byte array using the `ServiceOperation` class
6. Obtain the classification details of the component in XML format

When the required details are ready, the information is bundled together in the `CODSDetails` format. The transformation of data is based on the following steps:

1. Create a new instance of `CODSDetails`
2. Specify the information of the provider
3. Obtain the executables of the component from `ServiceOperation` class in the form of jarred byte array
4. Put the jarred byte array into `CODSDetails`
5. Put the serialized `TypedComponent` representation into `CODSDetails`
6. Put the serialized XML classification into `CODSDetails`
7. Encrypt the `CODSDetails` using the agreed DH key between the provider and the broker

The submitting of components to the broker is performed by invoking the `addService` method in the `ProviderSessionInterface` class. The `ProviderApp` class performs a check to determine the calling to `addService` or `modifyService` method in the `ProviderSessionInterface`. The check is achieved by querying the `DownloadedComponentList` to see if the component exists in the broker before invoking the method.

The package contains a `Security` class that is responsible for the management of keys in the application. The main methods in the class are `decrypting` and `startDHSession`. The `decrypting` method is used to decrypt the encrypted message in the `CODSDetails` obtained from the space using an agreed key generated from the `startDHSession` method.