3-26-1997

Object access control to enable internet commerce

Chengqiang Chen
Florida International University

FLORIDA INTERNATIONAL UNIVERSITY

Miami, Florida

OBJECT ACCESS CONTROL

TO ENABLE INTERNET COMMERCE

A thesis submitted in partial satisfaction of the
requirements for the degree of

MASTER OF SCIENCE

IN

COMPUTER SCIENCE

by

Chengqiang Chen

1997

To: Dean Arthur W. Herriott
College of Arts and Sciences

This thesis, written by Chengqiang Chen, and entitled OBJECT ACCESS CONTROL TO ENABLE INTERNET COMMERCE, having been approved in respect to style and intellectual content, is referred to you for judgment.

We have read this thesis and recommend that it be approved.

Nagarajan Prabhakaran

Chung-min Chen

Raimund K. Ege, Major Professor

Date of Defense: March 26th, 1997.

The thesis of Chengqiang Chen is approved.

Dean Arthur W. Herriott
College of Arts and Sciences

Dr. Richard L. Campbell
Dean of Graduates Studies

Florida International University, 1997

ACKNOLEDGEMENTS

I would like to thank my advisor, Dr. Raimund K. Ege for his guidance and patience throughout the course of my thesis.

Special thanks are also due to Dr. Nagarajan Prabhakaran, Dr. Chung-min Chen and Dr. Toby Berk for their helpful comments and guidance that have made my accomplishments possible.

My friends Victor Ortiz, Dr. Zhifeng Dong and Qing Liu provided me with their insightful feedback and discussions on drafts of the thesis. Mingjian Wang and Ming Zeng helped me with their invaluable ideas on programming. I appreciate their great help.

I would, also, like to thank Dr. Tih Yi and Dr. Fang Zhao for their generous support.

ABSTRACT OF THE THESIS

OBJECT ACCESS CONTROL TO ENABLE INTERNET COMMERCE

by

Chengqiang Chen

FLORIDA INTERNATIONAL UNIVERSITY, 1997

Miami, Florida

Professor Raimund K. Ege, Major Professor

Object-Orientation defines components (objects) that encapsulate data and functionality. Modern OO programming languages have features that specify the degree of encapsulation in much detail. This thesis extends the access control specification capabilities to objects and puts more emphasis on the objects of Internet commerce. A simple and flexible framework of access control is introduced based on the concept of “token”. A prototype implemented in Java demonstrates the feasibility of the ideas and related issues.

Table Of Contents

Table Of Figures

1 INTRODUCTION

The Internet explosion has led to the dramatic growth of *Internet commerce*. The Internet proved to be an effective place to conduct commerce due to various advantages it possesses. There are many success stories of doing business on the Internet. An Internet shopper using an Internet browser, such as Netscape Navigator, can order a PC or purchase online tax services. Both customers and sellers would like their commercial transactions conducted securely and safely. This leads to the problem of *access control.* [JECF96]

Object-Orientation defines components (objects) that encapsulate data and functionality. *Encapsulation* is the concept of packaging software components [PCW85] by specifying their interfaces to their clients and hiding their internal implementation details. Modern OO programming languages have features that specify the degree of encapsulation in much detail. By extending access control capabilities to objects, developers can control access to objects on the Internet in a general and flexible manner. This will be useful to Internet commerce.

This thesis extends access control specification capabilities from encapsulated members (methods, attributes) to objects, these objects are called *Service* objects here. A reusable framework based on a simple concept of "token" is established to enable access control over Internet commerce objects.

2 JAVA AND INTERNET COMMERCE

2.1 Internet Commerce

Electronic commerce, while developed many years ago on mainframe computers, recently took off with the explosion of the Internet. Now the Internet is the most important place to conduct electronic commerce. Driven by intense competition and cost-consciousness, businesses are evolving to Internet commerce. [KW96]

Internet commerce has different definitions when viewed in different perspectives. From the online perspective, Internet commerce can be defined as the transactions of information, products, and services conducted via Internet. Internet commerce enables new commerce channels anywhere within the World Wide Web. Internet commerce has become popular for the following reasons: [KW96]

- The Internet is available almost anywhere and anytime.
- The World Wide Web provides a user-friendly interface to the Internet and a standard, open model for delivering information.
- Transactions on the Internet can be faster and more accurate.
- Transactions are conducted directly between customers and sellers without necessity to meet; the process is interactive and the feedback from customers can be instant.
- Transactions are in real-time

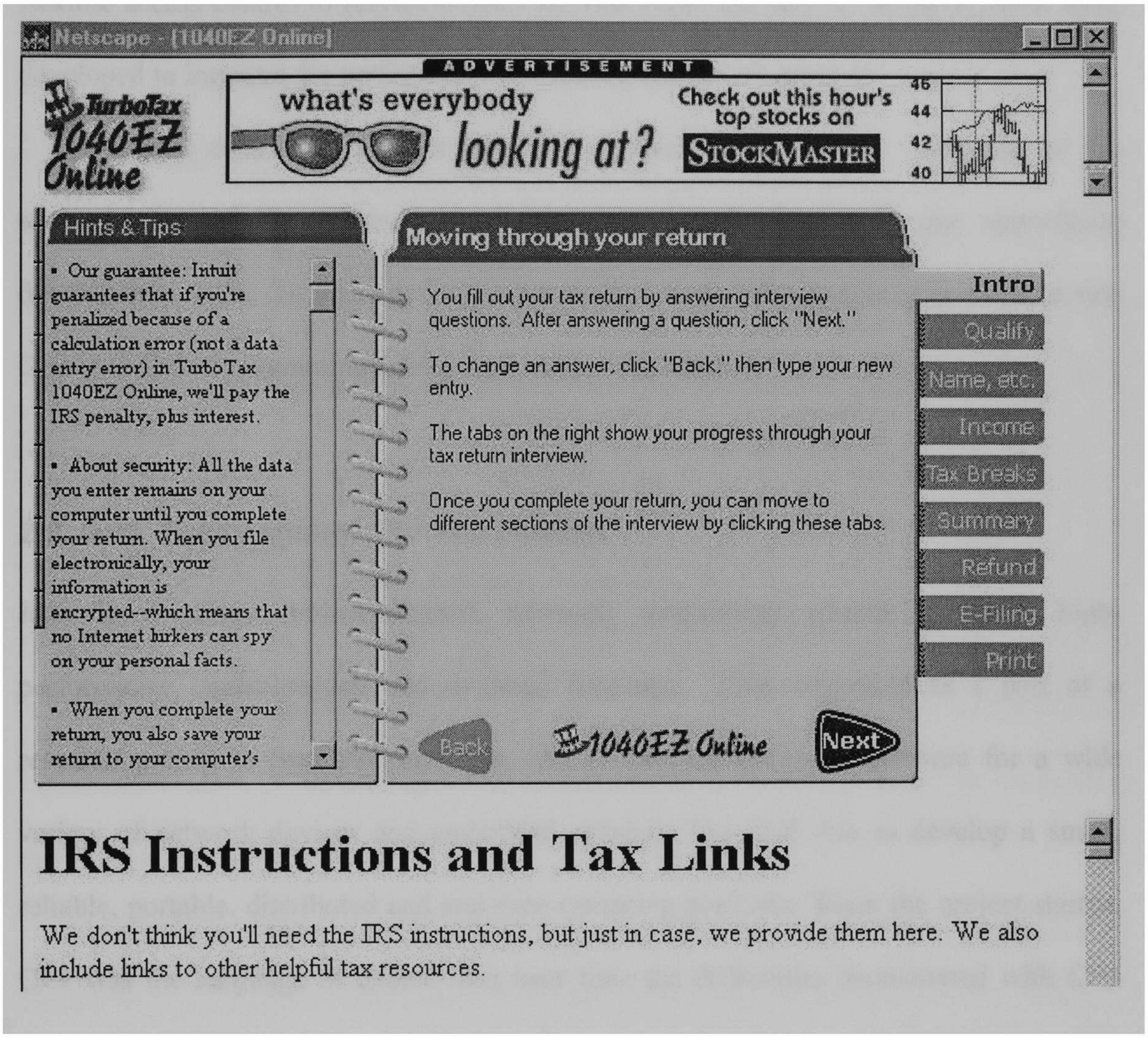
The primary focus of Internet commerce is on business-to-business transactions. Large businesses like banks are taking advantage of the Internet as the new medium to perform financial and merchant transactions. These transactions are in the area of supplier inventory and payment. Conducting transactions on the Internet can speed up processing, reduce transaction costs, increase sales, and improve customer services. Many of these improvements cannot be achieved by alternative methods with the same level of quality.

Large organizations often have complicated systems to maintain commerce activities inside these organizations, these activities include publishing corporate information, and coordinating corporate activities. These systems are called organization *Intranet*s. Since these systems often integrate with the Internet, we can include these activities in Internet commerce. Internet commerce is not only conducted between large organizations.

The third kind of Internet commerce is the most popular activity. It includes transactions such as online purchasing, online services and social management between corporations and consumers. From a simple perspective, there are two kinds of products best suited to online sales:

(1) Products that can be delivered over the Internet, such as information services or software. These sales best deliver the promise of Internet commerce. Software companies like Microsoft and Borland have Web sites to sell their software products. UPS provides an easy and efficient online package tracking service. Many banks have begun to provide online personal finance services. Figure 1 is a

Java applet "TurboTax 1040EZ Online" developed by Intuit that provides online tax services.



• Our guarantee: Intuit guarantees that if you're penalized because of a calculation error (not a data entry error) in TurboTax 1040EZ Online, we'll pay the IRS penalty, plus interest.

• About security: All the data you enter remains on your computer until you complete your return. When you file electronically, your information is encrypted--which means that no Internet lurkers can spy on your personal facts.

• When you complete your return, you also save your return to your computer's

You fill out your tax return by answering interview questions. After answering a question, click "Next."

To change an answer, click "Back," then type your new entry.

The tabs on the right show your progress through your tax return interview.

Once you complete your return, you can move to different sections of the interview by clicking these tabs.

IRS Instructions and Tax Links

We don't think you'll need the IRS instructions, but just in case, we provide them here. We also include links to other helpful tax resources.

Figure 1 An online tax service applet

(2) Products that may be sold in conventional stores, which can be advertised and/or ordered online to take advantage of the speedy, interactive benefits. For example, the Internet Shopping Network (http://www.internet.net/) offers online shopping for computer hardware like PCs, scanners.

Several issues must be addressed when discussing Internet commerce. Security and flexible access control to service objects are two important issues. Many systems were developed to improve the performance of security and access control.

The access control to Internet objects is studied in this thesis. Because of the sophistication of the Internet, it is important for developers to use appropriate development tools. To make developing software across Internet relatively easier, a new Object-Oriented programming language, Java, was created.

2.2 Java, The Language For The Internet

Java is a secure, object-oriented, network, architecture neutral, portable, high-performance, multithreaded and dynamic language. Java originated as a part of a research project at Sun Microsystems, Inc. to develop advanced software for a wide variety of network devices and embedded systems. The goal was to develop a small, reliable, portable, distributed and real-time operating platform. When the project started, C++ was the language of choice. But over time the difficulties encountered with C++ grew to the point where the problems could best be addressed by creating an entirely new language platform. Design and architecture decisions were drawn from a variety of languages such as Eiffel, SmallTalk, Objective C, and Cedar/Mesa. The result is a language platform that has proven ideal for developing secure, distributed, network-based end-user application environments ranging from network-embedded devices to the World-Wide Web, and to the desktop. [JAVA96]

2.2.1 Java Is Object-Oriented

Java is based on the best concepts and features of previous object-oriented languages, primarily Eiffel, SmallTalk, Objective C and C++. Everything in Java is either an object or able to be encapsulated within objects. Java is designed as closely to C++ as possible for most programmers that do object-oriented programming in C++. Java goes beyond C++ by omitting many confusing, poorly designed features of C++, such as pointers, operator overloading, multiple inheritance and extensive automatic coercion. Java also adds automatic garbage collection. Consequently, Java is easier and more efficient than C++. [JAVA96]

2.2.2 Java Is Designed For Networked/Distributed Environments

The explosive growth of the Internet and the World-Wide Web has lead to a complete new way of developing software. Java must be secure, high performance, and robust on multiple platforms in heterogeneous, distributed networks to enable electronic commerce and distributed computing.

Networks are composed of heterogeneous systems in general. To make applications execute anywhere on a network, the Java compiler generates byte code -- an intermediate format that is architecture independent, to transport code efficiently to multiple hardware and software platforms. The interpretative nature of Java solves both the binary distribution problem and the version problem; the same Java generated byte codes will run on any platform. Hence, Java is architecture neutral and portable. [JAVA96]

A lot of emphasis has been placed on security. Java enables the construction of virus-free, tamper-free systems. The authentication techniques are based on public-key encryption. With security features designed into the language and the run-time system, Java lets you construct applications that can't be invaded from the outside. In the network environment, applications written in Java are secure from intrusion by unauthorized code attempting to get behind the scenes and create viruses or invade file systems.

Java supports multithread programming. It has a sophisticated set of synchronization primitives that are based on the widely used monitor and condition variable paradigm. By integrating these concepts into the language (rather than exclusively within classes) they become much easier to use and, hence, more robust. Java has an extensive library to allow network programming. Java applications can open and access objects across the net via Uniform Resource Locators (URLs) with the same ease that programmers access a local file system.[JAVA96][JLS96]

2.2.3 An Ideal Language For Internet Commerce

Java is an ideal development language to meet the challenges of the Internet. The complete Java system includes several libraries of utility classes and methods useful to developers in creating multi-platform applications. Briefly, these libraries are: [JAVA96][JLS96]

Basic Java language classes	- java.lang
The Input/Output package	- java.io
The Java Utilities package	- java.util
The Abstract Window Toolkit	- java.awt

Two additional libraries form the Java Core API:[JAPI96]

The Network package	- java.net
The Applet package	- java.applet

Package java.applet defines Java applet. Java applets are programs which are downloaded from a server and run inside a Java enabled Web browser such as Netscape Navigator rather than a standalone Java application. Java applets are important to the support of Internet commerce.

The newly released JDK(Java Development Kit) 1.1 adds many new packages. The following four packages form the Java Enterprise API: [JDK1.1]

- Java Database Connectivity (JDBC). JDBC API is a standard SQL database access interface. This API provides Java programmers with a uniform interface to a wide range of relational databases, and provides a common base on which higher level tools and interfaces can be built. JDBC is now a standard part of Java and is included in JDK 1.1. [JDK1.1]
- Remote Method Invocation (RMI). RMI enables the programmer to create distributed Java-to-Java applications, in which the methods of remote Java objects can be invoked from other Java virtual machines, possibly on different hosts.

- Object Serialization. Object Serialization extends the core Java Input/Output classes. Object Serialization supports the encoding of objects and the objects reachable from them into a stream of bytes and it supports the complementary reconstruction of the object graph from the stream.
- Java Interface Definition Language (IDL). The Java IDL provides a way for transparently connecting Java clients to network servers using the industry standard IDL.

The JavaBeans APIs define a portable, platform-neutral set of APIs for software components. JavaBean components will be able to plug into existing component architectures such as Microsoft's OLE/COM/Active-X architecture, OpenDoc, and Netscape's LiveConnect. End users will be able to connect together JavaBeans components using application builders. For example, a button component could trigger a bar chart to be drawn in another component, or a live data feed component could be represented as a chart in another component.[JB96]

The Java Electronic Commerce Framework (JECF)API defines an architecture to support electronic commerce business transactions. The JECF, a virtual point-of-sale device implemented in software for use in Java-enabled environments, is a secure, extensible framework to enable Internet commerce.

The Java Core Reflection API is a special, type-safe, and secure API. It supports introspection about the classes and objects in the current Java Virtual Machine. The Core Reflection API is employed in out implementation.

Other APIs, such as, the Java Server API, the Java Security API, the Java Management API, the Java Media API, and the Java Embedded API are on the way. Java is still growing. From JDK 1.0 to 1.1 releases, the development system has expanded and some of the APIs have changed. The language itself is still evolving. These improvements make Java a more mature and powerful development platform, especially suitable for developing systems on the Internet.

3 ACCESS CONTROL

3.1 Encapsulation and Access Control Concepts

Access control is highly related to one of the key elements of Object-Orientation, *encapsulation.* Encapsulation is the concept of packaging software components by defining their interfaces to access clients and hiding their internal implementation details. [PCW85] There are several definitions for encapsulation, for the convenience of discussion, the following definition is applied:

Encapsulation is the collection and protection of the attributes of a concept and its exclusive, associated services. This concept is represented as a software component that collects and protects its information, its structure, and its services. Encapsulation can be thought of as the combination of abstraction and information hiding. [EGE96][NIE89]

A typical Internet software component, for example, a "Game" class which is used to offer an online game playing service (Figure 2), has two attributes ("players" and "scores"). It, also, has services to display information about the players, and two services ("begin" and "restart") to control the game. The degree of access control is some what arbitrary. Given a "game": are its attributes "players" and "scores" accessible? are its services "begin" and "restart" able to be invoked? How long can this game be? It is the purpose of the encapsulation specification to control the degree of invisibility or hiding of each software component. [EGE96]

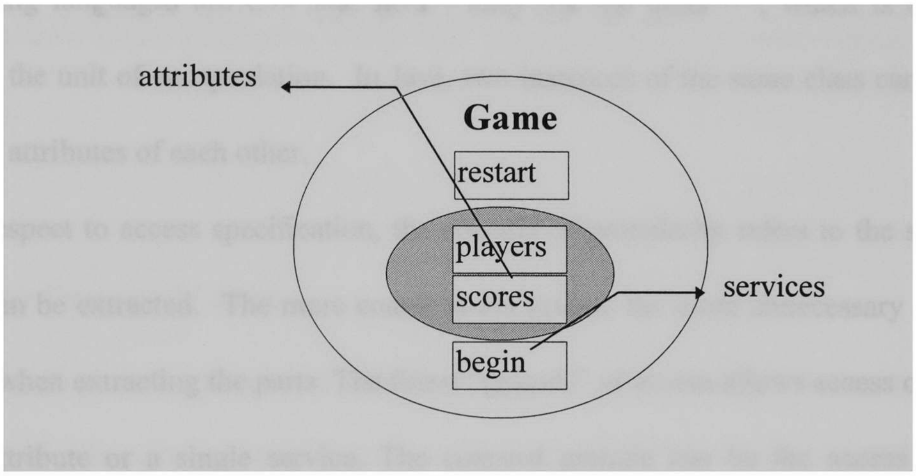


Figure 2 An encapsulated class: Game

The object-oriented community already has many ways and models for controlling access to elements of software components. However, how the visibility can be expressed in the most flexible way and how it can be effectively controlled are focused in this study. Our definition of *access control* is given as:

Access control is the specification of the visibility of elements of software components.

3.2 Encapsulation Unit and Access Granularity

An object is the most natural unit of encapsulation. Attributes and services are protected from access by outside objects. Most object-oriented programming languages including Smalltalk[GR83] adopt this model. For example, if an attribute of an object is "private", then it is not accessible to any other object. Exceptions among the object-oriented programming languages are C++ and Java. They use the class[ST91], which is a set of objects, as the unit of encapsulation. In Java, two instances of the same class can access the private attributes of each other.

With respect to access specification, the concept of granularity refers to the smallest unit that can be extracted. The more coarse is the access, the more unnecessary borders may exist when extracting the parts. The finest "granule" of access allows access of either a single attribute or a single service. The coarsest granule can be the access of "all services" or "all attributes". Java allows a per attribute/service specification of accessibility. For complex objects that contain other (or refer to other) objects, the contained objects can be viewed as conventional attributes of the container object. [EGE96]

Another important issue of encapsulation is the nature of the access. What effect does the access have on the state of the object? Access of an attribute is possible in an "initialization" sense (to set stored values for an attribute once, initially). It is also possible in a "read" sense (to get the stored value for an attribute) or in a "write" sense (to change the stored value). Access for a service can be read only, initialize, or write.

The access control specification capabilities will be extended to objects in this study. That is, an object can be the unit of access specification. This kind of specification includes complex objects that also contain other (or refer to other) objects. However, these contained objects can be viewed as attributes of the container.[EGE96]

This purpose of this study is to design a mechanism to implement access control in a flexible manner. The flexibility here can be understood as the degree of control, for example, if we divide the degree of access control from yes to no into 90 units, then we can specify how much freedom a *Client* may have to access a *Service* object, i.e., 30 units.

3.3 Access Control Of Java

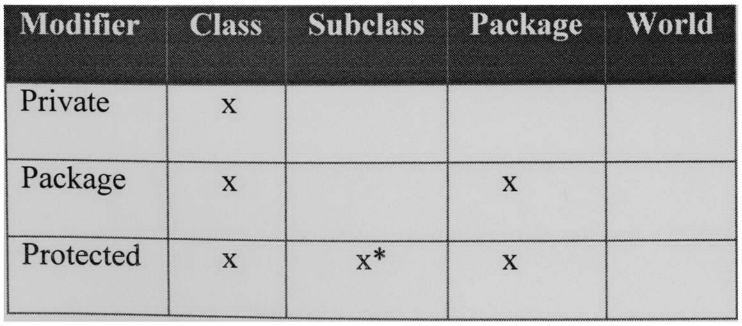
Access control in Java prevents the users of a package or class from depending on unnecessary details of the implementation of the package or class. Access control applies to the qualified access and to the invocations of constructors by class instance creation expressions, explicit constructor invocations, and the method *newInstance* of class *Class*. If access is permitted, then the accessed entity is said to be accessible.

Java access modes include three modes that come from C++: *private, protected* and *public.* Java add a new access mode: "*package*". Though they are similar in some way, the underlying object model of C++ does not actually enforce access modes. Unlike C++, every Java object can only be created by the *new* operation.[GATE97]

When a new class is declared in Java, the level of access permitted to its members is specified. As stated above, Java provides four levels of access. Three of them must be explicitly specified: *public, protected*, and *private.* Members declared public are available to all other classes. Members declared protected are accessible only to subclasses of that class, and nowhere else. Members declared private are accessible only from within the class in which they are declared and they are not available even to their subclasses.[JAVA96]

The fourth type of access is a special access level. It can be called "*package*" level access. It is the access level you obtain if you do not specify otherwise. The "*package*" access level indicates that the class members are accessible to all objects within the same package, but inaccessible to objects outside the package. Package is a useful tool for grouping together related collections of classes and interfaces.[JT96][JLS96]

Figure 3 shows the access levels permitted by these access modifiers.



Modifier	Class	Subclass	Package	World
Private	x			
Package	x		x	
Protected	x	x*	x	

Public	x	x	x	x

Figure 3 Access levels in Java

The first column indicates whether the class itself has access to the member modified by the access modifier. As you can see, a class always has access to its own members. The second column indicates whether subclasses (regardless of which package they are in) have access to the member. The third column indicates whether classes in the same package as the class (regardless of their parentage) have access to the member. The fourth column indicates whether all classes have access to the member.[JT96]

Note that the protected/subclass intersection has a superscript '*'. We need to discuss this a little further. For example, if we have classes A and A1, where class A1 is a subclass of A, then:

- If class A and A1 are in the same package, class A1 can access the protected members of class A.
- If they are not in the same package, A1 cannot access A's protected member.

3.4 Related Work

3.4.1 KAPSEL

The paper "Access Control Specification for Object-Oriented Software Components"[EGE96] introduces a specification language-KAPSEL to facilitate the specification of components. KAPSEL supports the concept of "key" access. When a KAPSEL class is specified, the specification may include a key, which is a set of access specifications per attribute or service. This key is given to the creator of the class; and the creator can access the new object according to the key, or can give the key to other objects. Keys can be copied, reduced, where access privileges to attributes and /or services can be removed. In order to gain access to an object via a key, the accessing object has to register with the object to be accessed (the creator object is automatically registered). This "key" concept allows very flexible, per object access specification of encapsulation control. It is also used in the implementation of KAPSEL encapsulation enforcement.

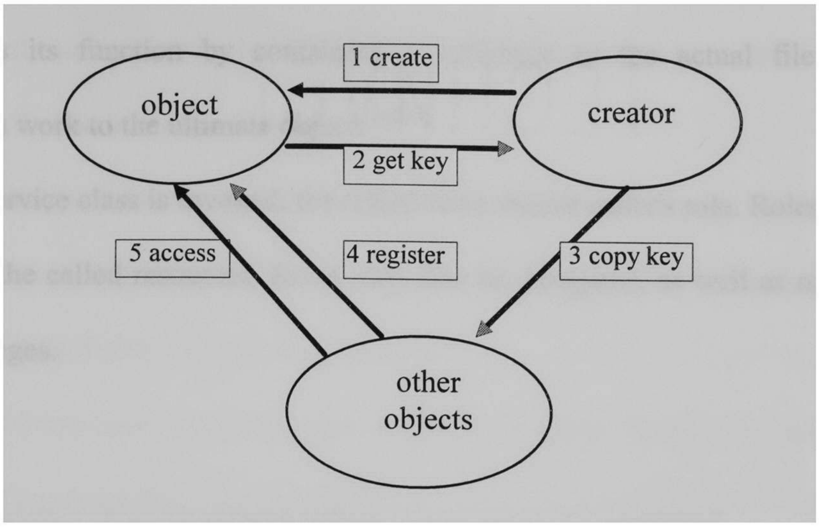


Figure 4 "Key" in KAPSEL

3.4.2 Sun's Java Electronic Commerce Framework (JECF)

The Java Electronic Commerce Framework (JECF) is Sun's framework for conducting business on the Internet. This thesis discusses what JECF provides for access control.

In JECF, access control is implemented in the Cassette Architecture by using the following: "roles"[VN], interfaces, two kinds of controlled access, and two flavors of objects. The Cassette Architecture is sometimes called a capability architecture. For instance, an object, such as one representing a disk file, needs controlling access to the file. Controlling access is needed, for example, to give certain users permission to read and write and giving other users only read permission. The Cassette Architecture accomplishes this by creating two new kinds of Permit objects. One provides read/write access to the file and the other provides read-only access. These objects are called Permits, or (historically) capabilities because they permit their possessors to invoke some of the methods defined for the ultimate objects they represent. The Permit object accomplishes its function by containing a reference to the actual file object and delegating its work to the ultimate object. [JE961]

When a Service class is invoked, the called class checks caller's role. Roles are mapped to rights on the called resources. Roles may also be delegated, as well as narrowing the access privileges.

4 "TOKENS" THAT ENFORCE ACCESS CONTROL

Both KAPSEL and JECF enforce well designed access control. JECF implements that by constructing a complex framework. While KAPSEL presents a more general and flexible way using the "key" conept The "key" is used to enable the access control[EGE96]. The "key" is given to the creator of an instance of the class. Other objects must get or copy the "key" from the creator if they want to access that object. Access control by "key" has some complexities: in order to gain access to an object via a key, an object should first register with the object that is to be accessed. How can an object invoke services of another object in flexible degree while keeping the process simple?

In this section, a new design to enforce the access control will be introduced.

As mentioned in section 3.3, Java provides four levels of access control over encapsulated elements, including attributes and methods. If a method is public, then it can be invoked by any object anywhere. Our system will be not restricted by these four levels, it will provide the degree of control according to the need of the service provider, i.e., we have these four access levels to define an outline of access control, then, we can provide more freedom of access control between them.

The granularity of the access control in Java is a single attribute or method. There is no access control over an object. We extend this concept and make the object be a control unit, i.e., a Service object on the Internet.

The unit of encapsulation is the class because we chose Java as the implementation tool. Java also provides the "class" object.

The specification of access control can be active or passive. Active specification controls which other objects are able to access specific attributes, methods or Service objects, e.g., the "friend" mechanism in C++.[ES90] Passive specification puts the attributes, methods and objects into two cases: the target is able to be accessed or not. Passive access specification is enabled in our design, and the active specification is only discussed in the section of future enhancements.

The access control in our project is implemented using the concept of a "token". It can be called "token-based access control". It is a simple and flexible mechanism that enables access control of Internet objects.

4.1 The "Token" Concept

The concept "token" is an extension of KAPSEL's "key" concept. It serves as the base of our "token-based access control". A token can be used to access the Service object or to specify the degree of access control. We define a *public defined token* as follows:

An object with *color, value* and other attributes. It is defined publicly and distributed by authorized objects.

A token is a kind of privilege. Tokens are used as the proofs of some degree of access authority to Service objects. A Service object publishes the requirements of access to its attributes, services or objects with tokens, i.e., a token with 10 green units for a game object. An object (Client in our project) that wants to access the game must get the proper tokens and submit them to the game (or other objects trusted by the game object).

The Service object checks the tokens when it receives the access request, then makes decisions on granting the access privilege or not.

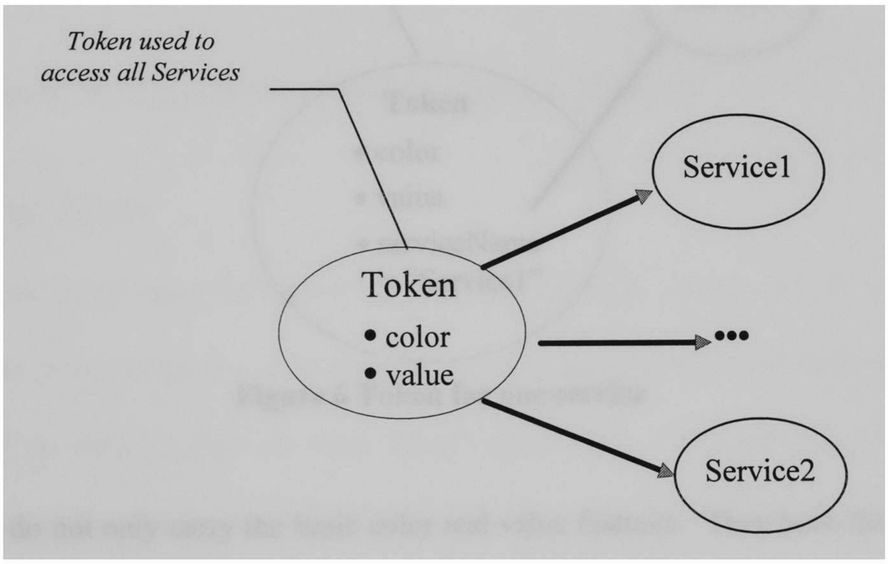


Figure 5 Token with color, value attributes for all services

Access tokens are issued by the Service object itself or authorized objects of that Service, like a Server. Only a Server can create and destroy tokens. Consequently, the Service objects cannot create or destroy tokens.

Public defined tokens from different places can interoperate. For example, a corporation that has two departments that provide Internet services. Tokens of one department for some Services can circulate in another department for other Services as long as the Servers of the two departments make proper exchanges. This exchangeability and color, value features make the "color token" a simple, flexible way to specify the degree of access control.

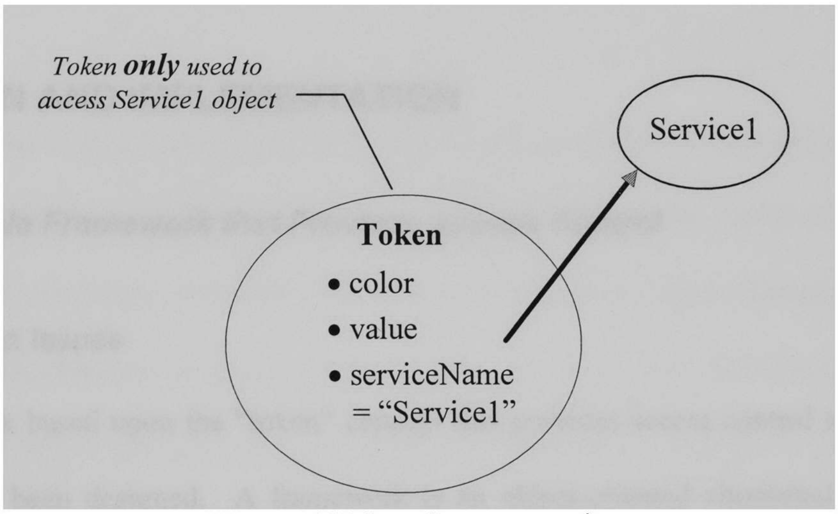

Token only used to access Service1 object
Service1
Token
• color
• value
• serviceName = "Service1"

Figure 6 Token for one service

Tokens do not only carry the basic color and value features. They have the option to carry the name of a Service object, a method or an attribute. The name must be unique within the context, i.e., in the format of class.methodName. Tokens that carry the target object name can only be used to access that object.

Another possible attribute that may be added to the token is the signature of the issuer to identify the source and restrict the circulation area of the token. If the Client, Service, and Server can authenticate the signature of the token, the security of the system can be improved greatly. The present implementation does not support this level of security, the signature is ignored.

Tokens have operations such as addition, subtraction. Only the same kind of tokens (with same color and serviceName) can do addition or subtraction operations.

The implementation of Token in Java is discussed in section 5.2.1, while the source code is listed in the Appendix A.

5 DESIGN AND IMPLEMENTATION

5.1 A Simple Framework that Provides Access Control

5.1.1 Design Issues

A framework based upon the "token" concept that provides access control to objects at runtime has been designed. A framework is an object-oriented abstracted design. It consists of an abstract class for each major component. The interfaces between the components of the design are defined in terms of a set of messages. As a part of the framework, there will usually be a set of subclasses that can be used as components in the design.[JF88][EGE92] The effort has been made to capture the essential elements of access control in Internet commerce, identify activity patterns, and construct a framework that encapsulates these features.

Two basic rules are followed in our design:

- Develop a small framework that provides a generic set of classes to implement access control. Future developers can extend those classes in the framework to describe the real problems of Internet commerce. [GHJV95]
- Keep the framework easy to implement and maintain. Make the reuse relatively simple and flexible.[PY91]

Consider an example of Internet commerce: A company plans to provide online game services via the Internet. The customers are people all over the world who have the proper access privileges. Customers send requests for games online. The server answers these requests, manages the games, and customers. Here, the game is the target of access control. The access permissions to the game are expressed in tokens, i.e., a token with 30 green units for a game and a token with 5 green units for the first step of the game. A customer's access privilege is expressed in the tokens that it possesses. The Server must determine if a customer has the correct tokens before granting a customer's request for a game.

Our project is to develop a set of reusable classes that enable the access control for objects of Internet commerce. These classes serve as a framework to provide "token-based access control" to objects, particularly, to those objects in Internet commerce, such as customer-ordered software packages or messages. We have added a monitor to observe and to modify the access specifications of objects visually. The project allows the user to do the following:

- gain access to objects. There is a "token" access specification for each object, attribute, or service to be accessed, e.g. for a chess game, 20 green units. Here, the objects to be accessed are instances of the GeneralService class or its subclasses. An object(Client) which wants to access a GeneralService object can get the access specifications from the GeneralService object or from a Server(an object that the GeneralService trusts). It dispatches requests for tokens to the Server. In this way, it gains access to the GeneralService object through the use of tokens.
- change access control. Given the initialization of access specification of a GeneralService, the Server can change the specifications of the access control at a later time, e.g., from 20 green units to 10 green units, or from 20 green units to 5 red units.
- monitor access A monitor is added to the framework to view the access control specifications dynamically and visually. Usually, the monitor is combined with the Server to get and set the access specifications.

5.1.2 Framework Outline

The research work is to implement a system as an extension to the Java language. The system consists of a set of reusable Java classes built on top of the Java APIs These classes are the following: the Token class, the Server class, the Service (GeneralService) class, the Client class, the Monitor class,as well as other classes.

Figure 7 shows the interaction diagram of the framework.

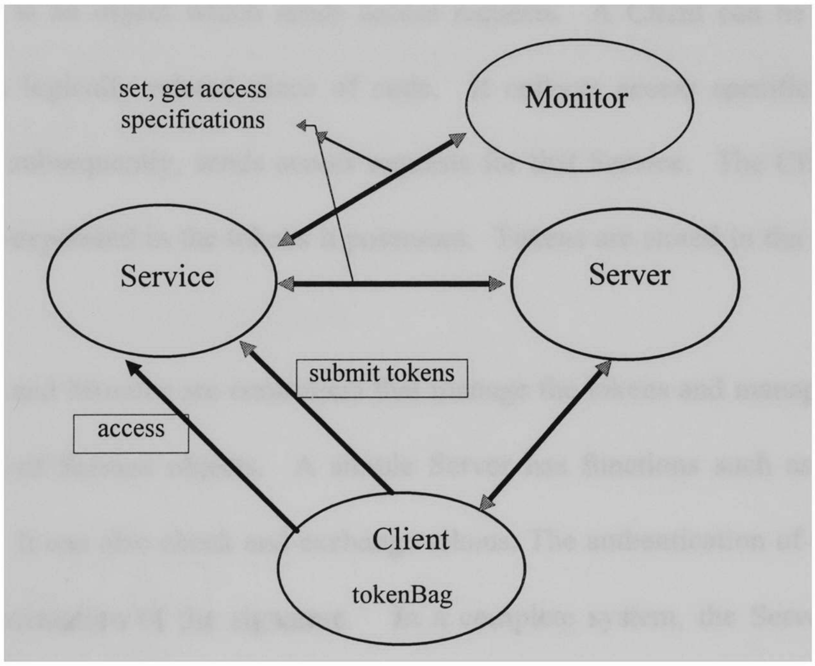


Figure 7 Token-based access control

The Service class plays an important role in the framework. It is the target of the access control. Objects can inherit behavior from the Service class in order to provide controllable services. In Java, the Service class is an object, as well as the attributes (fields), and the methods (services). Class Field, Method, and Constructor implement the

Member interface. Instances of these classes are created only by the Java Virtual Machine. These classes are used to manipulate the underlying objects, operate on field values, invoke methods on objects or classes, and create new instances of classes. [JDK1.1] Details of these classes are in the Java Reflection Model API of JDK version 1.1.

The Server, Client and other objects can get access specifications for a Service object. However, access requests for a Service are granted if the Client's tokens meet the access specifications.

The Client is an object which sends access requests. A Client can be a person, a program or a logically related piece of code. It collects access specifications for a Service, and, subsequently, sends access requests for that Service. The Client's access privileges are expressed in the tokens it possesses. Tokens are stored in the tokenBag of a Client.

The Server and Monitor are controllers that manage the tokens and manage the access specifications of Service objects. A simple Server has functions such as startServer, startServices. It can also check and exchange tokens. The authentication of tokens relies on the implementation of the signature. In a complete system, the Server must also manage Clients and Services. It has functions such as system policy setting, user management, token management, service dynamic management and security enforcement. The Server should be able to trace tokens and collect statistical information about tokens, Services and Clients. For example, to track what tokens a user currently has?

For Internet commerce, a custom Server can be the special server ready to provide services on the Internet.

The Monitor can get and set the access specifications for Service objects. In a simple implementation, the Server and the Monitor can be merged into one class that has one graphical user interface.

5.2 Classes

There are several important classes: Token, Server, GeneralService, Client and Monitor. They are described in this section.

5.2.1 The Token Class

The class definition of Token is given by:

```
class Token
{
   Color     color;
   int       value;
   String    serviceName; // The token can only used to access that service.

   //constructor
   Token();
   //(methods)
   add();
```

```
        sub();
        getColor();
        getValue();
        getServiceName();
        setServiceName();
        isSameKind();          //compare two tokens
    }
```

The class SpecUnit shown below is highly related to the Token class. The SpecUnit associates the degree of access control (expressed in a token) to a GeneralService object. The GeneralService object is identified by the serviceName. The serviceName of the SpecUnit must be unique within the context. Each SpecUnit object has an entry in the access control table.

```
class SpecUnit
{
    String   serviceName;
    Token    accessToken; //define color, value etc.
}
```

5.2.2 The GeneralService (Service) Class

The top of a class hierarchy should be abstract[JF88]. The GeneralService class is the super class of various objects to be accessed. In Internet commerce, goods and services are common subclasses of the GeneralService class. Each instance of GeneralService or its subclasses is created by a Server, or created under the permission of a Server, so a GeneralService instance can carry the signature of the Server to verify tokens. A GeneralService instance has a list of access specifications in tokens for attributes, methods, and the object itself. When the access requests from a Client arrive, the GeneralService object checks the tokens with the help of a Server; and grants the access requests if the access requests have been verified. For example, after submitting the proper tokens, a customer's request for playing a game is granted; and the customer can start the game.

Java did not provide a mechanism for multiple inheritance in order to avoid the problems brought by multiple inheritance. However, the features of multiple inheritance are important to express relations in the framework. As a result, the interface is used to simulate multiple inheritance in this study. Interfaces are used to define a protocol of behavior that can be implemented by any class anywhere in the class hierarchy. The limitation of interface lies in: [JT96]

1) It's impossible to inherit variable or method implementations from an interface
2) The interface hierarchy is independent of the class hierarchy--classes that implement the same interface may or may not be related through the class hierarchy. This is not true for multiple inheritance.

These limitations caused a lot more work in the implementation.

Below, the interface for GeneralService defines some common behavior of the GeneralService class:

```
interface GeneralServiceInterface
{
  abstract  SpecUnit  getAccessSpec();  //fetch access specification for a service
  abstract  boolean   addAccessSpec();
  abstract  boolean   setAccessSpec();
  abstract  boolean   grantService();
  abstract  int       checkToken();
 }


class GeneralService
 {
    String          serviceName;
    static Vector   accessSpec;       //array of SpecUnit

    //methods
    accessInit();                     //access specifications initialization
    getAccessSpec();
    setAccessSpec();                  //add or modify access specifications
    grantService();
```

```
        checkToken();
      }
```

For instance, a game is a kind of Service. The Game class implements GeneralServiceInterface, and defines some general operations of games like begin(), end() as well as the attribute, players.

```
 interface GameInterface extends GeneralServiceInterface
{
   public void   restart();
 }

class Game extends Applet implements GameInterface
/* derive from the Applet,  because applet is one of the most popular format of Internet
 *  games, and the applets can also run as applications in our predefined frames
 */
{
     UserID    players[];              //String
     int         score[];
     void        begin();
     void        end();
     ....
 }
```

5.2.3 The Server Class

Usually a Server is the creator of GeneralService instances. A GeneralService instance trusts the Server who creates it, that is, the GeneralService keeps the signature of the Server and accepts tokens carrying the signature of the Server. In our implementation, the signature is ignored because it is difficult to implement. The Server is responsible for distributing tokens to various Clients (within Internet commerce, customers usually buy tokens for some goods or services). It also authenticates and exchanges tokens.

In this implementation, the Server merges its functions with the Monitor to simplify the operations.

```
interface ServerInterface
{
   startServer();
   startServices();
   sellToken();             //send token to client according to client's request
   checkToken();            //authenticate the tokens
   exchangeToken();
}
```

5.2.4 The Client Class

Client is an object which sends requests to access to GeneralService objects. A Client must get the proper tokens to access a particular attribute, method(service) of an object. When a Client wants to access a GeneralService instance, it gets the access specification of that instance, sends access requests together with the necessary tokens to the

GeneralService object or the Server. If requests are granted, the Server or the system may return the desired service objects, and the services can be started. The cost of the access will be reflected in the changes of the tokens in the Client's tokenBag.

The tokens of a Client are stored in a container called the tokenBag. Tokens of the type have only one entry in the tokenBag. If new tokens of the same type are added, they will be merged into one entry.

The Client has some common operations. Client.importToken() is the method to process access privilege with Server and Monitor That is, how much authority the Server will give to the Client? This question is answered not by the Client, but by the Server based on the state of the Client, i.e., in an online sale instance, how much authority a customer may have often depends on how much money is left in his account. In another word, tokens are the forms of the access right.

The method Client.forService() consists of several steps: sending access requests, submitting access tokens and processing returned objects. First, two steps are needed to prepare to access the Service object. The Services requested include attributes, methods, and objects. If Client's access is for attributes and objects, the Client.forService() retrieves the objects. if the access is for methods, then the desired methods are invoked.

```
interface ClientInterface
{
    importToken();  //get the token from the server
    forService();   //send requests access objects, services and attributes
```

```
        }
```

The Client class implements the ClientInterface with ID, which is the Client identifier.

```
class Client implements ClientInterface
{
    String  ID;
    Vector  tokenBag;          //container of tokens
}
```

The User class is derived from the Client class and is a kind of Client with graphical interactive interface. Users may have password, if necessary. The UserFrame class used in the User class is a customized window. Java is designed to run on different platforms, therefore, the Client and the User class can be reused, customized under different environments and by different users.

```
class User extends Client
{
    String          Password;
    UserFrame   userFrame;  //UserFrame is the interactive frame

    //User();
    requestService();
    startService();
```

```
        showTokenBag();

        showGameBag();

    }
```

An instance of class User shown in Figure 11.

5.2.5 The Monitor Class

The Monitor class is an interactive tool of the controller of the system used to monitor the Services. The Monitor can set access specifications of the attributes, methods and objects of GeneralService. At runtime, the monitor can also change the specifications, i.e., the requirement to access the wave game is changed from 30 blue units to 10 red units. The Monitor is designed using the Java AWT packages. Hence, it can run on any platform where a Java Virtual Machine runs. Usually a Monitor comes together with the Server.

```
class Monitor

{

    //window elements here

    getAccessSpec();  //get the current access status of a GeneralService

    setAccessSpec();

    showAccessSpec();

}
```

Besides the important classes discussed above, many classes are developed to ensure the access control experiment. Many of them are interactive user interfaces, i.e., frames, dialogs.

These classes are illustrated in following the OMT (Object Modeling Technique) diagram(Figure 8).[Rum91]

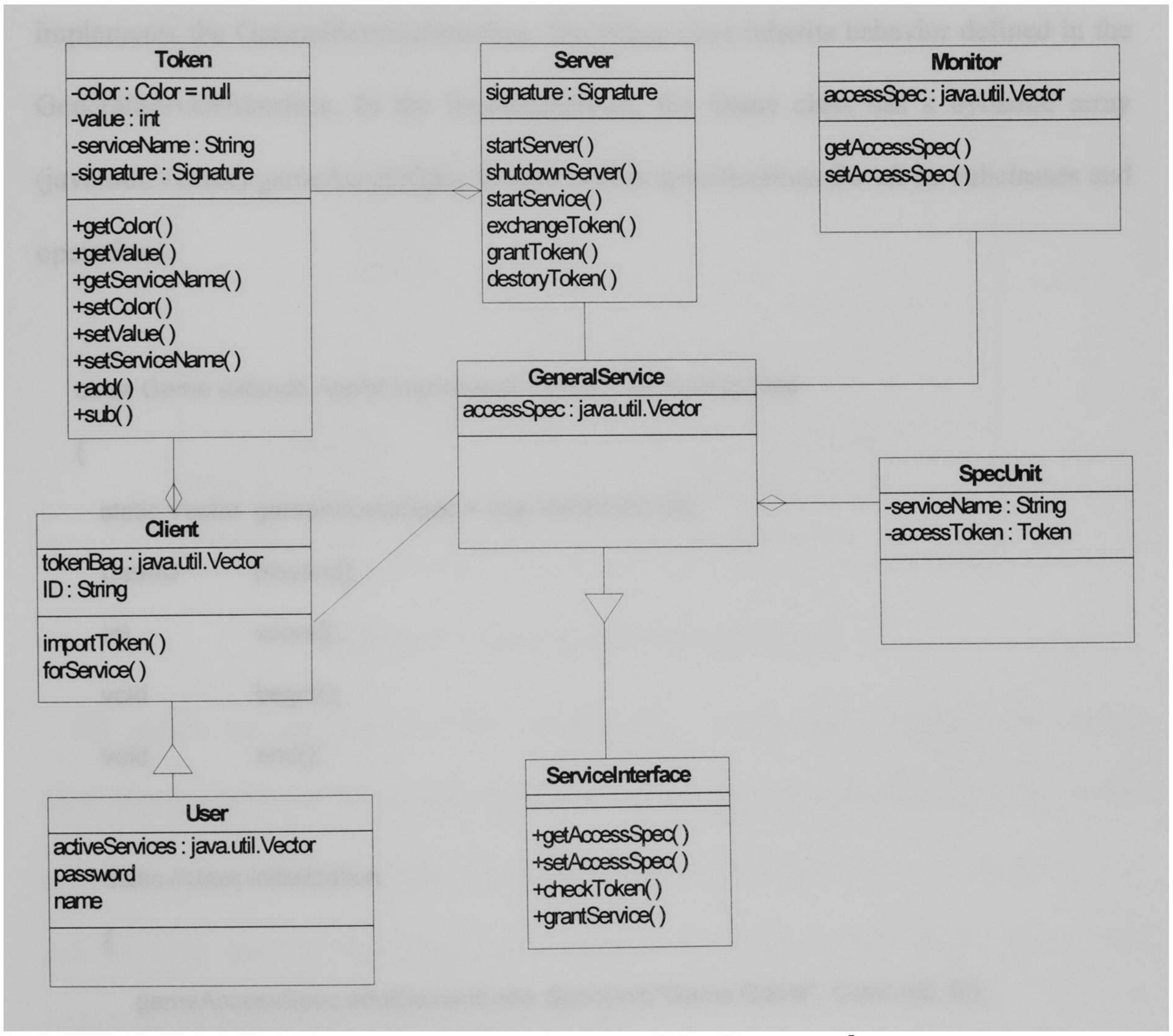


Figure 8 OMT diagram of the framework

5.3 Reuse Token-based Framework

5.3.1 Custom Service

To reuse the token access control framework within applications, first, we should define our own Service classes that are derived from the GeneralServiceInterface. For example, an Internet game zone needs access control to games, we can define the Game class that implements the GeneralServiceInterface. The Game class inherits behavior defined in the GeneralServiceInterface. In the implementation, the Game class has a dynamic array (java.util.Vector) gameAccessSpec to save access specifications for all its subclasses and operations.

```
class Game extends Applet implements GeneralServiceInterface
{
  static Vector  gameAccessSpec = new Vector(30,10);
  UserID         players[];
  int            score[];
  void           begin();
  void           end();

  static //class initialization
  {
     gameAccessSpec.addElement(new SpecUnit("Game.Game", Color.red, 0));
  }
```

```
    addAccessSpec();

    setAccessSpec();

    removeAccessSpec();

    ...

}
```

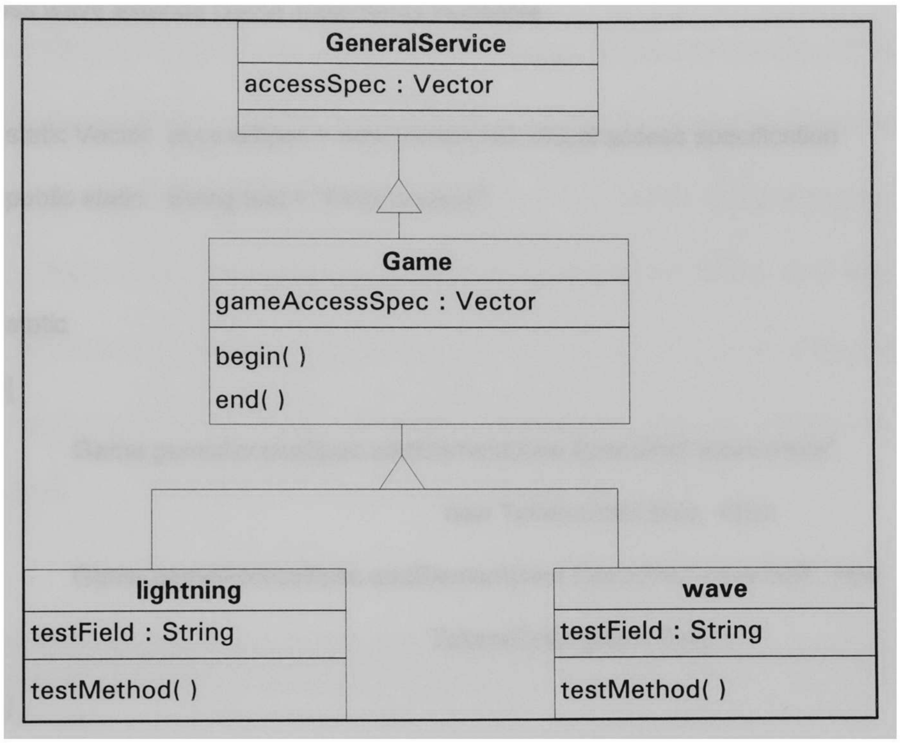


Figure 9 Class Game and its subclasses

All games are subclasses of the Game class. Every game registers the access specifications with the Game class. The lightning and wave games in the demo subsystem are subclasses of the Game class. From the sample code of lightning in the appendices, it can be seen that the lightning game registers its access specifications in its class initialization, which is done by this line of code:

```
Game.gameAccessSpec.addElement   (new   SpecUnit(   "lightning.lightning",   new
Token(Color.blue, 10)))



class wave extends Game implements Runnable
{
   static Vector  accessSpec = new Vector(10); //local access specification
   public static   String test = "Field in wave";

   static
   {
          Game.gameAccessSpec.addElement(new SpecUnit("wave.wave",
                                                  new Token(Color.blue, 10)));
          Game.gameAccessSpec.addElement(new SpecUnit("wave.test", new
                                                 Token(Color.green, 5)));
   }
}
```

5.3.2 Custom Classes: Client, Server and Monitor

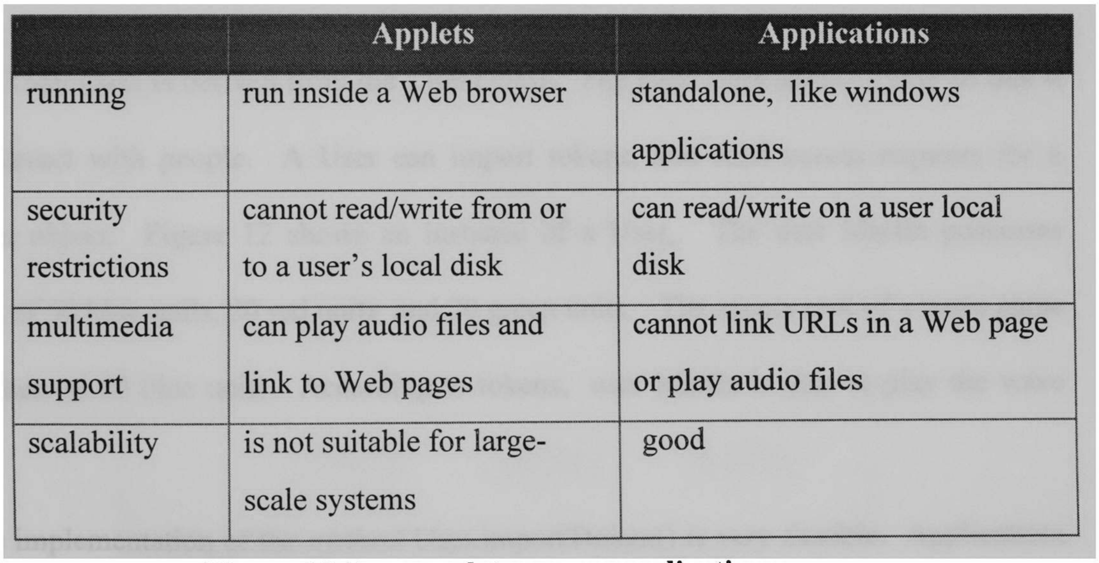
When the Service classes are ready, a custom monitor subclass is necessary. The monitor can gain and change access specifications.

Subclasses of Client and Server depend on the requirements of the systems. For example, if necessary, the Internet-ready client and server classes can be defined to take advantage of native network support within Java.

5.4 Test Package

The test package has custom Service class examples. These custom Service classes are several games that implement the GeneralServiceInterface. By setting parameters in the Monitor, the object access control experiments can be conducted visually with these games.

Java provides two ways to run a program. An applet, which runs inside a Web browser; or a standalone application, which can be invoked similar to Microsoft Windows or an X-Window program. For security reasons, applets cannot read or write from or to a user's local disk. Some tasks, like playing audio files or linking to URLs in a Web browser, cannot be done in an application; but an application can access a local disk, execute local programs, and much more. The comparison of applets with applications is shown in Figure 10.[FL96]



	Applets	Applications
running	run inside a Web browser	standalone, like windows applications
security restrictions	cannot read/write from or to a user's local disk	can read/write on a user local disk
multimedia support	can play audio files and link to Web pages	cannot link URLs in a Web page or play audio files
scalability	is not suitable for large-scale systems	good

Figure 10 Java applets versus applications

For the benefit of future development, the test package is developed as an application, though it can be run as an applet also. Because, currently, there is no Web browser support in JDK 1.1, we can only demo the applet with an early version developed under JDK 1.02. An early version of this applet demo is showed in Figure 11. Click the open button to start the demo.

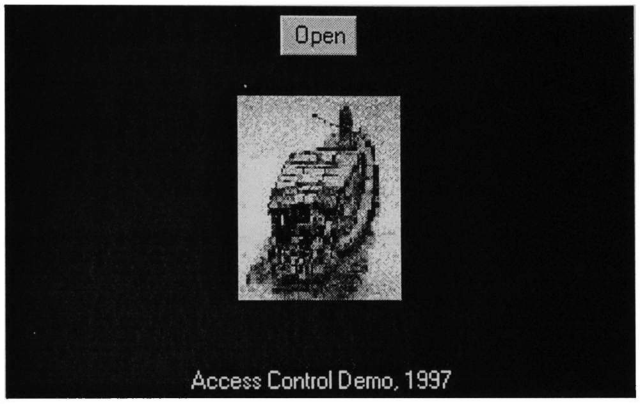
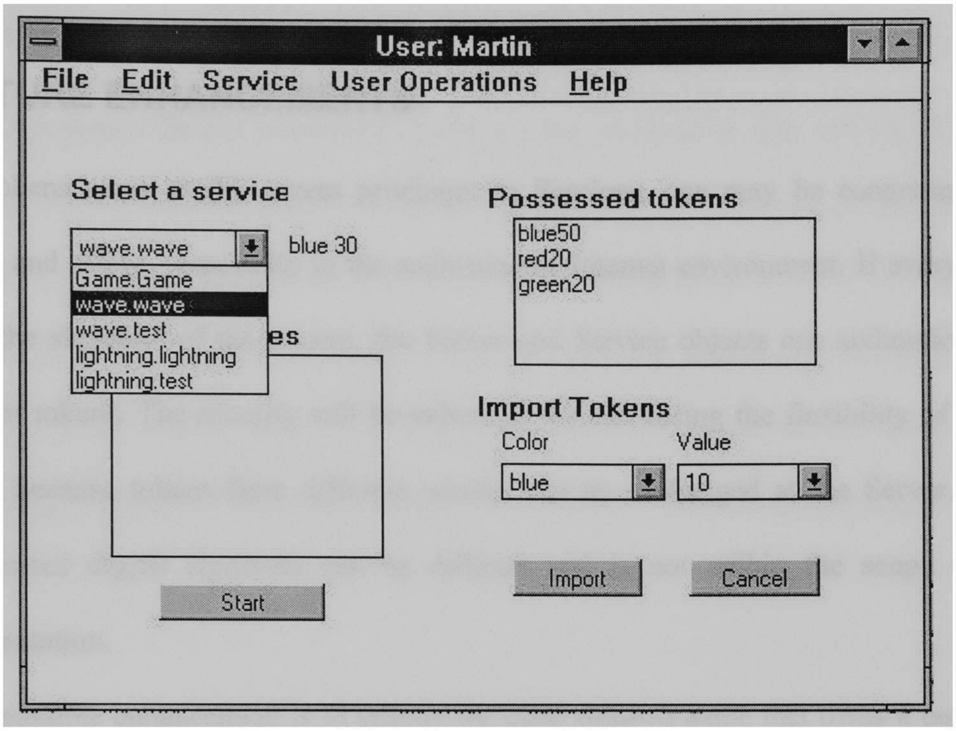


Figure 11 Start applet of the access control demo

The User class is derived from the Client class. The User class adds a frame so that it can interact with people. A User can import tokens, and send access requests for a Service object. Figure 12 shows an instance of a User,. The user Martin possesses tokens of 50 blue units, 20 red units and 20 green units. The access cost of a wave game is a token of 30 blue units. According to tokens, user Martin is able to play the wave game.

The implementation of the method User.importTtoken() is very flexible. Applications can define it according to the specifications of the system.



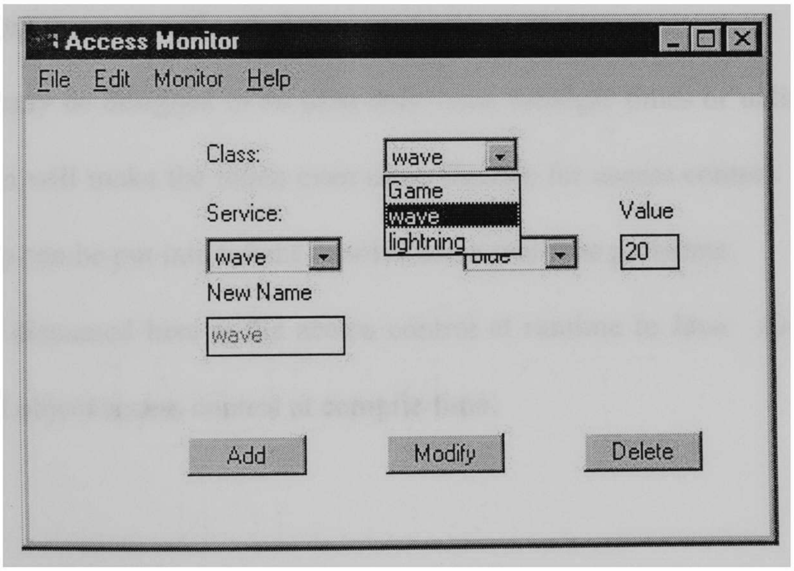
Select a service: wave.wave
Game.Game
wave.wave
wave.test
lightning.lightning
lightning.test
blue 30

Possessed tokens
blue50
red20
green20

Import Tokens
Color: blue
Value: 10

Figure 12 An instance of class User

Figure 13 is an instance of the Monitor class. It monitors the access specifications of the Services. The access specifications of these Services can be set or modified.



Class: wave
Game
wave
lightning
Service: wave
New Name: wave
Value: 20

Figure 13 Monitor

6 FUTURE ENHANCEMENTS

Since tokens serve as the access privileges to Services, one may be concerned with security and safety, especially in the sophisticated Internet environment. If every token carries the signature of the issuers, the Server and Service objects can authenticate the source of tokens. The security will be enhanced without losing the flexibility of access control, because tokens from different servers can be exchanged at the Server. Well-implemented digital signature can be difficult and is not within the scope of this implementation.

One possible enhancement is to specify the class (object) name that owns a particular token. Actually, the enhancement would be to add the unique ID of the Client class. In this case, the token can be used only by this Client to access Service objects. Active encapsulation control is actually enabled. An object can issue tokens, which allow other objects (Clients) to see specific attributes, methods, or objects.

The token may be designed to be used only once, multiple times or unlimited number of times. This will make the token even more flexible for access control. Furthermore, the time stamp can be put into tokens to solve some real time problems.

The model discussed here is the access control at runtime in Java. Another issue is how to extend object access control at compile-time.

7 SUMMARY

Object-Orientation defines components (objects) that encapsulate data and functionality. Modern OO programming language have features that specify the degree of encapsulation in much detail. This thesis extends the access control specification capabilities to objects, and make objects controllable to various access requests. By introducing tokens as a kind of general virtual money, degrees of the access control to objects can be specified. A prototype implemented in Java demonstrates the feasibility of the ideas and related issues.

REFERENCES

[EGE92]Raimund K. Ege and Christian Stary. "Designing maintainable, reusable interface". IEEE Software, November 1992.

[EGE96]Raimund K. Ege. "Access Control Specification for Object-Oriented Software Components". School of Computer Science, Florida International University, 1996

[ES90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*, Addison-Wesley, 1990.

[FL96] David Flanagan. *Java in a Nutshel.,* O'Reilly & Associates, Inc, 1996.

[GATE97]Javasoft, Sun, "The Gateway Security Model in the Java Electronic Commerce Framework", 1997.

[GHJV95]Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[GR83] Adele Goldberg and D.Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley Reading, Mass, 1983.

[JAVA96]JavaSoft, Sun, "The Java Language Environment, A White Paper", 1996, http: //java.sun.com /doc/language_environment/.

[JAPI96]James Gosling and Frank Yellin. *The Java Application Programming Interface, Volume I: Core Package.* Addison-Wesley, 1996.

[JB96] JavaSoft, Sun, "Java Beans 1.0", 1996, http://splash.javasoft.com/beans/.

[JDK1.1]Javasoft, Sun, "Java Development Kit API version 1.1", 1997.

[JE961] JavaSoft, Sun , "Writing Code for the JECF", 1996, http://java.sun.com/commerce/.

[JECF96]JavaSoft, Sun. "White Paper: The Java Electronic Commerce Framework (JECF)", 1996, http://java.sun.com/commerce/.

[JF88] Ralph E. Johnson, Brian Foote. "Design Reusable Classes", Journal of Object-Oriented programming, June/July 1988, pp22-35.

[JLS96] JavaSoft, Sun "The Java Language Specification 1.0", 1996.

[JT96] Mary Campione and Kathy Walrath. *The Java Language Tutorial: Object-Oriented Programming for the Internet*, Reading, MA, Addison-Wesley, 1996.

[KW96] Ravi Kalakota, Andrew Whinston. *Frontiers of Electronic Commerce.* Addison-Wesley, 1996.

[NIE89] Oscar Nierstrasz. "A Survey of Object-Oriented Concepts", *Object-Oriented Concepts, Databases and Applications*, ed. W.Kim and F. Lochovsky, pp.3-21. ACM Press and Addison Wesley, 1989

[PCW85]David L. Parnas, Paul C. Clements, and David M. Wesis. "The modular structure of complex systems". *IEEE Transactions on Software Engineering, SE-1193*, March 1985.

[PY91] Coad, Peter and Ed Yourdon. *Object-Oriented Design.* Englewood Cliffs, NJ, Prentice Hall, 1991.

[Rum91]James Rumbaugh et al, *Object-Oriented Modeling and Design*, Prentice Hall, 1991.

[ST91] B. Stroustrup. *The C++ programming Language.* Addison-Wesley, 1991.

[VN] Michael VanHilst and David Notkin. "Using Role Components to Implement Collaboration-Based Designs". Department of Computer Science and Engineering, University of Washington, Seattle, Washington.

(The source code of the "token-based access control" framework has seven files. We put two files as appendices.)

Appendix A: GeneralService.java

```
/**
 * GeneralService.java                          Last updated 2/3/1997
 *
 * This file defines the basic classes in token-based access control framework:
 *                  token, specUnit, GeneralService and interface GeneralServiceInterface
 * These classes serve as basic blocks in the framework.
 * Custom classes can be derived from these basic classes and interfaces.
 */
import java.lang.*;
import java.awt.Color;
import java.util.Vector;


/**
 * Token: base of access control, tokens act as the proof of the access authority.
 * Token can carry serviceName to specify name of the service to be accessed
 */
class Token
{
//Signature signature;              //used to restrict the circuration area;
String serviceName = null;          //token can only be used to acccess that service;
Color  color = null;
int    value = 0;

  //constructors
  Token(String serviceName, Color color, int value )
  {
     this.serviceName = serviceName;
     this.color = color;
     this.value = value;
  }

  Token(Color color, int value)
  {
     this.serviceName = null;
     this.color = color;
     this.value = value;
  }

  Token(Color color)
  {
   this.serviceName = null;
   this.color = color;
   this.value = 1;
}

/*Add tok.value to this token, return a new created token with the sum.
 *Return null if add cannot processed for it's not the same type of token
 */
Token add(Token tok)
{
  if (this.isSameKind(tok))
  {
   this.value += tok.value;
   return (new Token(this.color, this.value));
  }
   return null;
}

/* Sub tok.value from this.token.
 * Return null if          (1) not same type
 *                         (2) this smaller than tok(not only on value, color may count too)
 */
Token sub(Token tok)
{
  if (this.isSameKind(tok))
  {
   if (this.biggerEqual(tok))
   {
      this.value -= tok.value;
                 return (new Token(this.color, this.value));
   }
  }

  //cannot sub
  return null;
}

public String getServiceName()
{
        return this.serviceName;
}

public Color getColor()
{
        return this.color;
}

public int getValue()
{
        return this.value;
}
```

```
public void setServiceName()
{
        this.serviceName = serviceName;
}

public void setColor(Color color)
{
        this.color = color;
}
public void setValue(int value)
{
        this.value = value;
}

/* compare fields serviceName and color */
public boolean isSameKind(Token tok)
{
        if (this.color.equals(tok.color) )
        {
                if ((this.serviceName == null) && (tok.serviceName != null))
                        return false;
                else if ((this.serviceName != null) && (tok.serviceName == null))
                        return false;
                else if ((this.serviceName == null) && (tok.serviceName == null))
                        return true;
                else if (this.serviceName.equals(tok.serviceName))
                        return true;
        }
        return false;
}

/* Compare tokens;
 * Return  true if it has bigger value(and same color, serviceName) than tok
 */
public boolean biggerEqual(Token tok)
{
    if (this.isSameKind(tok))
    {
       if (this.value>=tok.value)
          return true;
    }
    return false;
}

/*like toString(), returns a string reflect color, value, and serviceName*/
public String tokenString()
{
    Integer val = new Integer(this.value);
    if (serviceName == null)
         return new String(this.color.toString()+" "+val.toString() );
    else
         return new String(this.serviceName+" "+this.color.toString()+val.toString() );
```

```
  }
}


/**
 * SpecUnit is used to define the degree of access control;
 * SpecUnit links a token and a service to form an item of access specifications
 */
class SpecUnit
{
String  memberName; //serviceName
Token   accessToken; //define degree in color, value and may serviceName

   public SpecUnit(String memberName, Token accessToken)
   {
      this.memberName  = new String(memberName);
      this.accessToken = accessToken;
   }

   public SpecUnit(String memberName, Color color, int value)
   {
      this.memberName  = new String(memberName);
      this.accessToken = new Token(color, value);
   }

   public SpecUnit(String memberName, String tokenServiceName, Color color, int value)
   {
      this.memberName  = new String(memberName);
      this.accessToken = new Token(tokenServiceName, color, value);
   }

   public String specUnitString()
   {
      Integer val = new Integer(accessToken.value);
      return new String(" " + memberName + " " + accessToken.tokenString());
   }
}


/**
 * This interface defines the behavior of all service classes (The importance lies in the
 * fact that Java doesn't support multiple inheritance)
 * To implement access control, every Service class extends from this interface.  The access
 * specifications initialized in static method accessInit() in class initialization;
 */
interface GeneralServiceInterface
{
          public void   begin();
          public void   end();

          //addAccessSpec in class initialize
          public boolean  addAccessSpec(String serviceName, Color color, int value);
          public boolean    setAccessSpec(String serviceName, Color color, int value);

          //Not set in token because signature may consider in future
          public boolean    setAccessSpec(String serviceName, String tokenServiceName,
                            Color  color, int value);
          public SpecUnit   getAccessSpec(String serviceName);
          public boolean    grantService();
          public int        checkToken();
}


/**
 *Class GeneralService; an example of root class.
 *To extend the object access control specification capabilities to object at runtime
 */
public class GeneralService extends Object implements GeneralServiceInterface
{
String serviceName;                 //optional
static Vector accessSpec;           //array of SpecUnits

   //class initialze
   static
   {
           accessSpec  = new Vector(10, 10);
   }

   public static void accessInit()     //not defined yet
   { ;}

   public void begin()                 //to start a service
   {
           ;                           //default setting
   }

   public void end()
   {
        ;                              //default setting to windows destruction
   }

   //constructor
   public GeneralService(String serviceName)
   {
      this.serviceName = new String(serviceName);
   }

   /*
    * Add an entry of access control for a member, an object
    */
   public static boolean addAccessSpec(SpecUnit setUnit)
   {
          accessSpec.addElement(setUnit);
          return true;
}

public boolean addAccessSpec(String serviceName, Color color, int value)
{
        accessSpec.addElement(new SpecUnit(serviceName, color, value));
        return true;
}

public boolean setAccessSpec(String serviceName, Color color, int value)
{
        return setAccessSpec(new SpecUnit(serviceName, color, value));
}

//Set service only accepts a special tokens
public boolean setAccessSpec(String serviceName, String tokenServiceName,
                        Color color, int value)
{
        return setAccessSpec(new SpecUnit(serviceName, tokenServiceName, color, value));
}

//replace or add an entry
public boolean setAccessSpec(SpecUnit setUnit)
{
   SpecUnit unit;

   if (setUnit == null )
      return false;
   if (accessSpec.size()<1)
   {
      accessSpec.addElement(setUnit);
      return true;
   }

   //search for the memberName, if found, replace it
   for (int i=0; i<accessSpec.size(); i++)
   {
      unit =  (SpecUnit) accessSpec.elementAt(i);
      if (unit.memberName.equals(setUnit.memberName))
      {
         try{accessSpec.setElementAt(setUnit,i);}
         catch (ArrayIndexOutOfBoundsException e) {;}
         return true;
      }
   }

   //add it if cannot find the member(service)
   accessSpec.addElement(setUnit);
   return true;
 }

 //public boolean setAccessSpec(String memberName,Color color, int value){}
 //return the access specification of the unit that have the memberName
```

```
  public SpecUnit getAccessSpec(String memberName)
  {
          SpecUnit theUnit;
          if (accessSpec.size()<1)
                  return null;
          for (int i=0; i<accessSpec.size(); i++)
          {
                  theUnit =  (SpecUnit) accessSpec.elementAt(i);
                  if (theUnit.memberName.equals(memberName))
                  {
                          return theUnit;
                  }
          }

          //not found, return null
          return null;
          }


  /* Two methods following will be overloaded if server need run in
   * real TCP/IP enviroment
   */
  public boolean  grantService()
  {
    return true;
  }

  //authenticate tokens
  public int     checkToken()
  {
    return 0;
  }
}


/* Examples of extending the generic class
class TaxService extends GeneralService
{
}
*/
```

Appendix B: User.java

```
/**
 * User.java   Last updated 2/2/1997
 *
 * This file defines class Client and User in "token-based" access control framework.
 * Notice:
 * The event models in JDK 1.02 and JDK 1.1 has great difference. This project
 * kept the model in JDK 1.02 for compatible reasons.  Another reason is there
 * is only JDK 1.1 beta available when this project finished.
 */
import java.lang.*;
import java.lang.reflect.*;
import java.awt.*;
import java.applet.*;
import java.util.*;
import GameFrame;
import lightning;      //two examples of Services: lightning, wave as games
import wave;
import UserFrame;
import Game;
import Token;
import SpecUnit;
import Monitor;

/**
 * ClientInterface: generic client interface
 *
 * Access clients must implement this interface to server Client in
 * Token-based framework
 */
interface ClientInterface
{
        public boolean importToken(Color color, int value); // get tokens from server (or other Clients)
        public Object  forService(String serviceName);          // for a service object
}


/**
 * Class Client, generic client;
 * Expanding version with ID, tokenBag and basic client operations
 */
class Client extends Object implements ClientInterface
{
String  ID;                  //Client ID; only in class User we need password
Vector  tokenBag;            //container of tokens

  //public Client(String ID)
  //{
  //   this.ID = new String(ID);
  //}
```

```
/* Merge tokens of same type in the tokenBag to avoid redundant
 * search, add second of one type to the first,
 * then set second value to 0, color to black for later delete
 */
protected void tokenBagMerge()
{
int    i = 0, j = 0, s = 0;
Token  t1, t2, t3;

  //merge the tokens of same color, signature
          s = this.tokenBag.size();
          if (s<2)  return;
          for (i=0; i<s-1; i++)
          {
             t1 = (Token ) tokenBag.elementAt(i);
             for (j=i+1; j<s; j++)
             {
                t2 = (Token) tokenBag.elementAt(j);
                        if (t1.isSameKind(t2))
                        {
                           //if (t2.value>=0) //not necessary
                           {
                              t3 = new Token(t1.color, t1.value+t2.value);
                              tokenBag.setElementAt(t3, i);
                              tokenBag.setElementAt((new Token(Color.black, 0)), j);
                           }
                        }
                     }
          }

          //remove the tokens marked with black color and 0 value
          s = tokenBag.size();
          for (j=0; j<s; j++)
          {
             t1 = (Token) tokenBag.elementAt(j);
             if ((t1.color.equals(Color.black)) && (t1.value==0))
             {
                        tokenBag.removeElement(t1);
                        s--;
             }
          }
}

  //public boolean importToken(String serviceName, Color color, int value) // get token from the server
  public boolean importToken(Color color, int value)
  {
             /*1 send request; assume request granted
              *2 receive tokens; merge token will be done later
              *return true  if import request is not granted
              */
             {
```

```
        this.tokenBag.addElement(new Token(color, value));
        tokenBagMerge();
        return true;            //success
            }
}

//get the second part (member) part (no class. prefix) of full member name in class.member format
public String memberName(String serviceName)
{
String memberName = null;

            StringTokenizer st = new StringTokenizer(serviceName);
            if (! st.hasMoreTokens())
                        return null;                        //empty
            st = new StringTokenizer(serviceName, ".");
            int i = 0;
            while (st.hasMoreTokens())
            {
                        if (i == 1)                         //one after "."
                                    memberName = new String(st.nextToken());
                        i++;
            }
            return memberName;
}

/*process request access for a service(member and object), manage client tokenBag
 *parameters: serviceName: class.member(to access object use class.class construct)
 *return: 1) constructor object for object access
 *        2) method object for method access //have access modifier restriction
 *        3) attribute object for attribute access
 *notice: memberName, no class prefix
 */
public Object forService(String  serviceName)
{
Class       cls  = null;
Object      obj  = null;
int         numOfParameter = 0, i;
String      className = null, memberName = null;      //not necessary for attribute name
SpecUnit unit = null;
Token       tok  = null;

            StringTokenizer st = new StringTokenizer(serviceName); //get name
            if (! st.hasMoreTokens())
                        return null;                        //empty name
            String fullName = new String(st.nextToken()); //maybe two parts:class.method...
            st = new StringTokenizer(fullName, ".");
            numOfParameter = 0;
            while (st.hasMoreTokens())
            {
                        if (numOfParameter == 0)
                                    className = new String(st.nextToken());
                        else if (numOfParameter == 1)
```

```
                memberName = new String(st.nextToken());
            else
                return null;                    //error name with
            numOfParameter++;
    }

    /* process */
    if (className == null)
            return null;
    try
    {
      cls = Class.forName(className);
    }
    catch (ClassNotFoundException e)
    {
      return null;
    }

    if (numOfParameter == 2)
    {
     if (className.equals(memberName)) //access object
     {
            Constructor[] cons = cls.getConstructors(); //no parameters
      if (processCost(serviceName) == false)
            return null;
      try{return (cons[0]);} //must have one
            catch(ArrayIndexOutOfBoundsException e) { return null;}
     }
     else //access member
     {
      /*Method:getMethods, get public methods, no need of parameter (no para!)
       *      getMethod,  methods must be public, require parameter
       *      getDeclareMethods, must be not inherited, no para, all accessible
       *Field: getFields, for public accessible fields
       *              getField(STring name), public !!!
       *      getDeclaredFields, no tinherited, no para, all accessible
       */
            /*Method[] method = cls.getMethods();
            for (i=0; i<method.getLength(); i++)
            {
                    if (method[i].getName().equals(memberName)) //no dupplicated name
                    {
                            return method[i];
                    }
            }
            */
            Field field = null;
            try
            {
              field = cls.getField(memberName);
            }
            catch (NoSuchFieldException e)
```

```
                    {
                      return null;
                    }
                    if (processCost(serviceName)!= false)
                            return field;
              }
            }
            return null;
    }

    /*
     * substract cost fromthe token bag
     * return: true: success; false: fail
     */
    public boolean processCost(String serviceName)
    {
    SpecUnit  unit = null;
    Token     tok  = null;
    Game      game = new Game();

            unit = game.getAccessSpec(serviceName);
            if (unit == null)
                    return false;
            return subCost(unit.accessToken);
    }

    //sub the cost
    //return false if it is not right operation
    public boolean subCost(Token costToken)
    {
            Token    tok = null;

            //substract cost tokens. note if the costToken.value == 0, then, access is granted
            if (costToken.value<=0)
                    return true;
            tokenBagMerge();
            for (int i=0; i<tokenBag.size(); i++)
            {
                    tok = (Token) tokenBag.elementAt(i);
                    if (tok != null)
                    {
                      if (tok.biggerEqual(costToken))
                      {
                        tok.sub(costToken);
                        return true;
                      }
                    }
            }
            InfoDialog d = new InfoDialog(new Frame()," ", "User didn't have enough tokens.");
            d.setVisible(true);
            return false;
     }
```

```
}

//User is the Frame-based Client
public class User extends Client
{
 String    password;
 protected String reqGameID;   //current game, necessary?
 UserFrame userFrame = null;
 Vector       activeGameBag;    //activeServices in OMT diagram, specify games related to this game

 public User(String ID, String password)
 {
          this.tokenBag  = new Vector(10, 10);
          this.activeGameBag = new Vector(10,10);
          this.ID        = new String(ID);
          this.password  = new String(password);
          this.userFrame = new UserFrame(this);

          //give the instance to register the class and services
          wave w1 = new wave();       //must create so the class initialize
          lightning l1 = new lightning();

          //add access specifications for object and members
          for (int i=0; i<Game.gameAccessSpec.size(); i++)
          {
            SpecUnit unit = (SpecUnit) Game.gameAccessSpec.elementAt(i);
            userFrame.serviceChoice.add(unit.memberName);
            userFrame.serviceChoice.setBounds(10,40,100,80); //re-setbounds due to bug in JDK 1.1 AWT
          }
          userFrame.setTitle("User: "+this.ID);
          userFrame.setVisible(true);
 }

 //this method may be put in the class Client to check
 //reuqest a service, check service exist, tokens and  authority
 public boolean requestService(String ID, String serviceName)
 {
          if (serviceName.equalsIgnoreCase("lightning")||serviceName.equalsIgnoreCase("wave"))
          {
                    return true;
          }
          else
             return false;
 }


  //override Client.forService
  public Object forService(String serviceName)
  {

    /*if (requestService(ID, serviceName)==true)
    {*/
```

```
        return super.forService(serviceName);
    /*}
    else
        return null;*/
}

//start the required service
public boolean startService(Object robj)
{
Game              game    = null;
GameFrame         frame   = null;
int               gameColor= 0;
boolean           rval    = false;
Class             cls     = null;
Object            obj     = null;

        if (robj == null)   return false;
        if (robj instanceof Constructor)
        {
                cls = ((Constructor) robj).getDeclaringClass();

        }else if (robj instanceof Field)
        {
                cls = ((Field) robj).getDeclaringClass();
        }
        else if (robj instanceof Method)
        {
                cls = ((Method) robj).getDeclaringClass();
        }

        try{obj = cls.newInstance();}
        catch(InstantiationException e){return false;}
        catch(IllegalAccessException e){return false;}

        //refresh possessedTokenList in UserFrame
        showTokenBag();     //tokenBag already substracted

        //start services(games)
        if (obj instanceof lightning)
        {
                game   = (lightning) obj;
                rval = true;
        }
        else if (obj instanceof wave)
        {
                game   = (wave) obj;
                rval = true;
        }
        else
                return (rval=false);

        if (robj instanceof Constructor)
```

```
                    else if (color.equals(Color.blue))
                              colorStr = new String("blue");
                    else if (color.equals(Color.black))
                              colorStr = new String("black");
                    userFrame.possessedTokenList.add(colorStr+String.valueOf(tok.value));
          }
}
public void showActiveGameBag() //in the possessedTokenList of UserFrame
{
Class   cls = null;
Object  obj = null;

          //clear old list, and add possessed
          userFrame.startedServiceList.removeAll();
          for (int i=0; i<activeGameBag.size(); i++)
          {
                    if ( (obj = (Object) activeGameBag.elementAt(i)) == null ) //not game, object is enough
                              return;
                     if (obj instanceof Constructor)
                     {
                      cls = ((Constructor) obj).getDeclaringClass();
                      userFrame.startedServiceList.add(cls.getName()); //give the service name
                     }
                     else if (obj instanceof Field)
                     {
                              cls = ((Field) obj).getDeclaringClass();
                       userFrame.startedServiceList.add(cls.getName()+"."+((Field) obj).getName());
                     }
                     else if (obj instanceof Method)
                     {
                               cls = ((Method) obj).getDeclaringClass();
                               userFrame.startedServiceList.add(cls.getName()+"."+((Field) obj).getName());
                     }
                     if (cls==null)
                               return;

  }
 }

 //main entry
 public static void main(String[] args)
 {
  String userName1 = new String("Martin");

          User user1= new User(userName1, "111");
          user1.importToken(Color.blue, 30);
          user1.showTokenBag();
          Frame  monitor = new Monitor();
          //monitor.setBackground(Color.blue);
          monitor.setVisible(true);
  }
 }
```

```
//utility dialog
class YesNoDialog extends Dialog
{
public static final int NO     = 0;
public static final int YES    = 1;
public static final int CANCEL = -1;

protected Button yes = null, no = null, cancel = null;
protected MultiLineLabel label;

  public YesNoDialog(Frame parent, String title, String message,
         String yes_label, String no_label, String cancel_label)
  {
    super(parent, title, true);
    this.setLayout(new BorderLayout(15, 15));
    label = new MultiLineLabel(message, 20, 20);
    this.add("Center", label);

    Panel p = new Panel();
    p.setLayout(new FlowLayout(FlowLayout.CENTER, 15, 15));
    if (yes_label != null) p.add(yes = new Button(yes_label));
    if (no_label != null)  p.add(no = new Button(no_label));
    if (cancel_label != null) p.add(cancel = new Button(cancel_label));
    this.add("South", p);
    this.pack();
  }

  // Handle button events by calling the answer() method.
  public boolean action(Event e, Object arg)
  {
    if (e.target instanceof Button)
    {
      this.setVisible(false);
      this.dispose();
      if (e.target == yes) answer(YES);
      else if (e.target == no) answer(NO);
      else answer(CANCEL);
      return true;
    }
    else return false;
  }

  //
  protected void answer(int answer)
  {
    switch(answer)
    {
      case YES:          yes();    break;
      case NO:           no();     break;
```

```
        case CANCEL:     cancel(); break;
      }
    }

    protected void yes() {}
    protected void no() {}
    protected void cancel() {}
}

class ReallyQuitDialog extends YesNoDialog
{
    //TextComponent status;
    public ReallyQuitDialog(Frame parent) //, TextComponent status)
    {
      super(parent, "Really Quit?", "Really Quit?", "Yes", "No", null);
      //this.status = status;
    }
    public void yes() { System.exit(0); }
    public void no()
    {
      ;//if (status != null) status.setText("Quit cancelled.");
    }
}


class InfoDialog extends Dialog
{
    protected Button button;
    protected MultiLineLabel label;

    public InfoDialog(Frame parent, String title, String message)
    {
      super(parent, title, false);
      this.setLayout(new BorderLayout(15, 15));
      label = new MultiLineLabel(message, 20, 20);
      this.add("Center", label);

      button = new Button("Okay");
      Panel p = new Panel();
      p.setLayout(new FlowLayout(FlowLayout.CENTER, 15, 15));
      p.add(button);
      this.add("South", p);

      this.pack();
    }

    // Pop down the window when the button is clicked.
    public boolean action(Event e, Object arg)
    {
      if (e.target == button) {
        this.setVisible(false);
        this.dispose();
      return true;
    }
    else return false;
  }

  public boolean gotFocus(Event e, Object arg)
  {
    button.requestFocus();
    return true;
  }
}
```