A Computer Aided Despatch System on Java/CORBA Platform

Progress Report

by

Chau Chi Wing

April 2000

Abstract

In this report, the design and implementation of a Computer Aided Despatch (CAD) system on Java/CORBA platform is described. CAD systems, like many other mission critical systems, were traditionally implemented on mainframe computers, and increasingly on client/server platforms. Mainframe computers have excellent reliability, but the capital investment and maintenance cost are both very high. Client/server platforms provide fancy user interface at relatively low capital investment, but the maintenance cost can be even higher than those of mainframe computers, due to the many system management issues of the fat clients. In this project, a 3-tier CAD system is implemented on Java/CORBA platform, with the client running as Java applet inside a web browser, and the server objects and RDBMS running on separate computers. The goal is to demonstrate the viability of using Java and CORBA to built the next generation enterprise-ready systems which are easy to maintain, extend, customize and interface with other systems

Contents

1 Introduction

Computer Aided Despatch (CAD) systems are used in the service industry to provide prompt service to their clients. The nature of services provided varies greatly, ranging from taxi-calling service, electric maintenance service, and the like, to emergency services and the military. The CAD systems deployed are mission critical as their failure could cause customer dissatisfaction, financial lost, or even lost of lives.

Even nowadays, many such systems still run on mainframe computers, with text-based terminals as the user interface. Their survival is partly due to their large size, which implies high cost for replacement. The other reason is their proven track record, in line with the high reliability of mainframe computers. However, the maintenance cost of these systems is very high, because their hardware is proprietary and personnel with mainframe expertise is becoming rare.

Some newer CAD systems are implemented as 2-tier client-server systems. These systems have fancy GUI and is reasonably reliable. However, they are difficult to manage due to their fat client architecture, which requires installation and periodic upgrading of many drivers and programs in many PC clients. Another problem is that they are not designed for wide distribution; if they are used in wide area networks, their performance suffers.

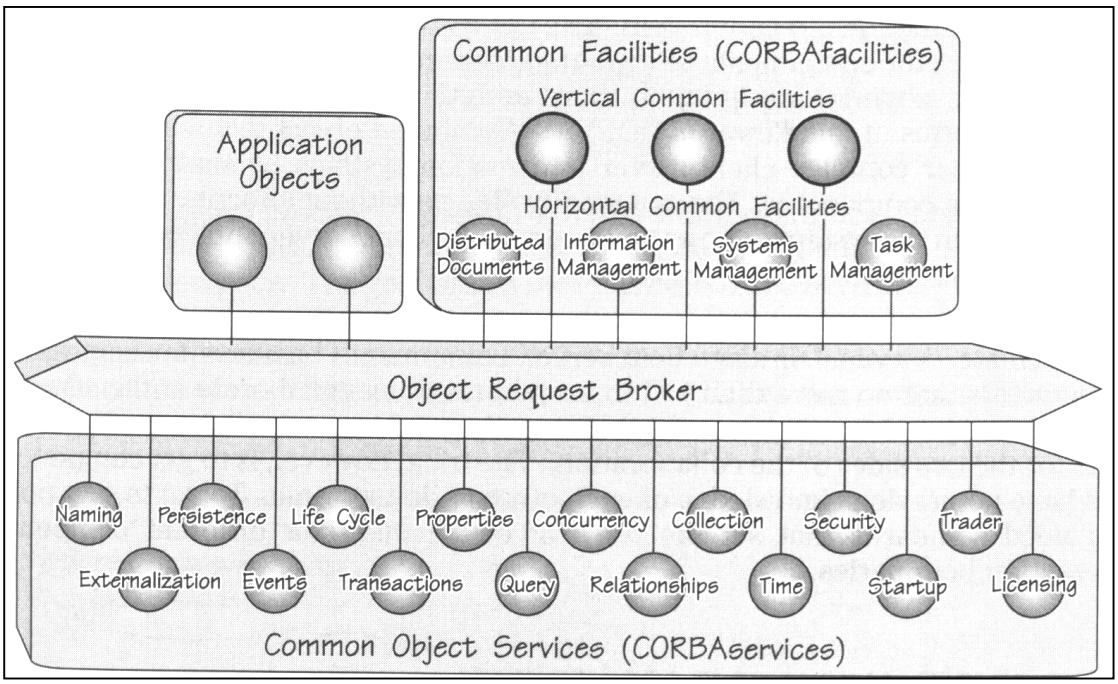
In this project, a baseline CAD system will be implemented on Java/CORBA platform. The goal is to built a system that is easy to maintain, extend, customize and interface with other systems and can be used over both intranets and the Internet. With appropriate customization, this system can be adapted to other specific environments.

The main purpose of this project is to demonstrate the use of Java and CORBA to build enterprise-ready system. Various useful concepts and mechanism, like Java applet, drag-and-drop (DnD) mechanism, CORBA and JDBC will be demonstrated.

2 Common Object Request Broker Architecture

2.1 Object Management Architecture

Figure 1. The OMG Object Management Architecture.



[2] The Common Object Request Broker Architecture (CORBA) is the most important middleware project ever undertaken by the computing industry. It is the product of the Object Management Group (OMG), which includes over 800 companies, representing the entire spectrum of the industry. The CORBA object bus defines the shape of the components that lives within it and how they interoperate.

CORBA is important because it defines middleware that has the potential of subsuming every other form of existing client/server middleware. At the same time, it provides a solid foundation for a component-based future.

CORBA was designed to allow intelligent components to discover each other and interoperate on an object bus. However, CORBA goes beyond just interoperability. It also specifies an extensive set of bus-related services for creating and deleting objects, accessing them by name, storing them in persistent

stores, externalizing their states, and defining ad hoc relationships between them.

CORBA support inheritance, that means you can create an ordinary object and then make it transactional, secure, lockable, and persistent by making the object multiply-inherit from the appropriate services.

In the fall of 1990, the OMG first published the Object Management Architecture Guide (OMA Guide). It was revised in September 1992. The details of the Common Facilities, however, were added later in January 1995. Figure 1 shows the four main elements of the architecture: 1) Object Request Broker (ORB) defines the CORBA object bus; 2) CORBAservices define the system-level object frameworks that extend the bus; 3) CORBAfacilities define horizontal and vertical application frameworks that are used directly by business objects; and 4) Application Objects are the business objects and applications - they are the ultimate consumers of the CORBA infrastructure. The following sections provide a top-level view of the most fundamental element – the ORB.

2.2 Object Request Broker

The Object request broker (ORB) is the object bus which allows objects to transparently make requests to and receive responses from other objects located locally or remotely. The client is not aware of the mechanisms used to communicate with, activate, or store the server objects. The CORBA 1.1 specifications introduced in 1991 specified the Interface Definition Language (IDL), language bindings and APIs for interfacing to the ORB. CORBA 2.0 specifies interoperability across vendor ORBs.

A CORBA ORB provides a wide variety of distributed middleware services. The ORB lets objects discover each other at run time and invoke each other's services. An ORB is much more sophisticated than alternative forms of client/server middleware, including the traditional Remote Procedure Calls (RPCs), Message-Oriented Middleware (MOM), database store procedures, and peer-to-peer services.

Benefits that every CORBA ORB provides:

Static and dynamic method invocations - A CORBA ORB allows developers to statically define method invocations at compile time, or dynamically discover them at run time. Hence, developers can get strong type checking at compile

time or maximum flexibility associated with late (run-time) binding. On the contrary, most other forms of middleware only support static bindings.

High-level language bindings - A CORBA ORB allows developers to invoke methods on server objects using their own choice. CORBA separates interface from implementation and provides language-neutral data type that make it possible to call objects across language and operating system boundaries. In contrast, other types of middleware typically provide low-level, language-specific, API libraries. Also they do not separate implementation from specification. The API is tightly bound to the implementation, which makes it very sensitive to changes.

Self-describing system - CORBA provides run-time metadata for describing every server interface known to the system. Every CORBA ORB supports an Interface Repository (IR) that contains real-time information describing the functions a server provides and their parameters. The clients use the metadata to discover how to invoke services at run time. It also helps tools to generate code on-the-fly. The metadata is generated automatically either by an IDL-language precompiler or by compilers that know how to generate IDL directly form an OO language, e.g. Visigenic/Netscape's Caffeine generates IDL directly from Java bytecode. CORBA is the first and also the most mature middleware to provide this type of run-time metadata and language-independent definitions of all its services.

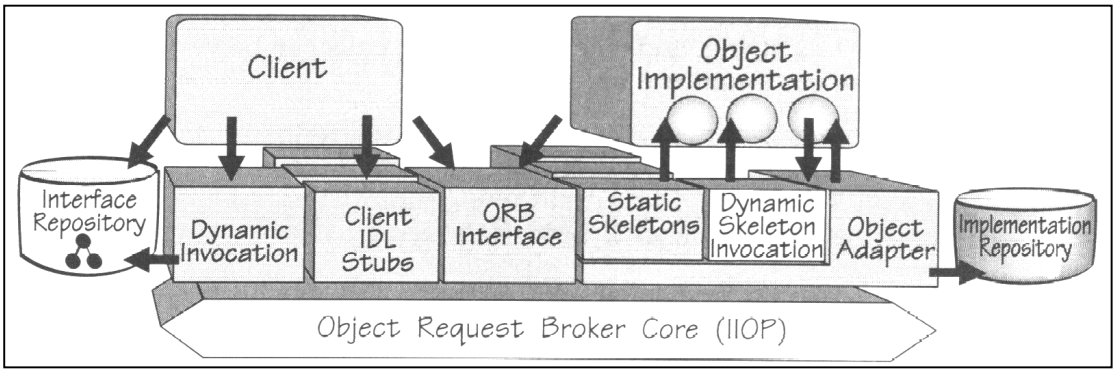
Location transparency - An ORB can run in standalone mode on a laptop, or it can be interconnected to every other ORB in the universe using CORBA Internet Inter-ORB Protocol (IIOP). An ORB can broker inter-object calls within a single process, multiple processes running within the same machine, or multiple processes running across networks and operating systems. This is completely transparent to objects

Built-in security and transactions - The ORB includes context information in its messages to handle security and transactions across machine and ORB boundaries.

Polymorphic messaging - In contrast to other forms of middleware, an ORB does not simply invoke a remote function. It invokes a function on a target object, which means that the same function call will have different effects, depending on the object receives it.

Coexistence with existing systems - CORBA's separation of an object's definition from its implementation is perfect for encapsulating existing application. Using CORBA IDL, a developer can make his own existing code look like an object on the ORB, even if it is implementation in stored procedures. This enables CORBA an evolutionary solution.

Figure 2. The Structure of a CORBA 2.0 ORB.



2.3 Anatomy of a CORBA 2.0 ORB

Figure 2 shows the client and server sides of a CORBA ORB. The client does not have to be aware of where the object is loaded, its programming language, its operating system, or any other system aspects that are not part of an object's interface.

The client IDL stubs provide the static interfaces to the object services. These precompiled stubs define how clients invoke corresponding services on the servers. From a client's perspective, the stub acts like a local call. It is a proxy for a remote server object. The stub perform marshaling so that the operations and the parameters are encoded and decoded into flattened message formats to send to the server.

The Dynamic Invocation Interface (DII) allows the discovery of methods to be invoked at run time. CORBA defines standard APIs for looking up the metadata that defines the server interface, generating the parameters, issuing the remote call and getting back the results.

The Interface Repository APIs let developers obtain and modify the descriptions of all the registered component interfaces, the methods they support, and the

parameters they required. CORBA calls these description *method signatures*. The Interface Repository is a run-time distributed database that contains machine-readable versions of the IDL-defined interfaces. The APIs allow components to dynamically accessed, tore, and update metadata information. This pervasive use of metadata allows every components that lives on the ORB to have self-describing interface.

The ORB Interface consists of a few APIs to local services that may be of interest to an application. For example, CORBA provides APIs to convert an object reference to a string, and vice versa. These calls can be very useful if the object reference is to be stored or communicated.

The Server IDL Skeletons provide static interfaces to each service exported by the server. These skeletons, like the stubs on the client, are created using an IDL compiler.

The Dynamic Skeleton Interface (DSI), introduced in CORBA 2.0, provides a run-time binding mechanisms for servers that need to handle incoming method calls for components that do not have IDL-based complied skeletons. The Dynamic Skeleton looks at parameter values in an incoming message to figure out the target object and method. In contrast, normal compiled skeletons are defined for a particular object class and expect a method implementation for each IDL-defined method. Dynamic Skeletons are very useful for implementing generic bridges between ORBs. They can also be used by interpreters and scripting languages to dynamically generate object implementation. The DSI is the server equivalent of a DII. It can receive either static or dynamic client invocations.

The Object Adapter sits on top of the ORB's core communication services and accepts request for service on behalf of the server's objects. It provides the run-time environment for instantiating server objects, passing requests to them, and assigning them object references. The Object Adapter also registers the classes it supports and their run-time instances with the Implementation Repository (below). CORBA specifies that each ORB must support a standard adapter called the Basic Object Adapter (BOA). Servers may support more than one object adapter.

The Interface Repository provides a run-time repository of information about the classes a server supports, the objects that are instantiated, and their IDs. It also

serves as a common place to store additional information associated with the implementation of ORBs, including trace information, audit trails, security and other administrative data.

The ORB Interface consists of a few APIs to local services that are identical to those provided on the client side.

3 Distributed Systems on CORBA

3.1 City of Pittsburgh – Crime Information System

[7]For Police, Fire, and other emergency personnel, historic information about the addresses that they are servicing can be lifesaving. For example, if the address in question has been visited 4 times in the past 30 days for domestic disputes, possible drug dealings, or recovery of a stolen weapon, emergency personnel would know to use an extra degree of caution. In Pittsburgh, this type of information has traditionally been kept in separate departments, divisional databases and servers across the city. Without easy access to this valuable information, emergency personnel had no way to know what dangers they might be facing when responding to a call.

Previously, to obtain crime and address history information, the police commander had to call City Information Systems (CIS) to submit a request. CIS would then write queries to extract the appropriate data from several disparate data sources, including 4 different databases on 4 different servers. These data sources stored Mayor's Service Center complaint calls, police records, building permits information, and 911 calls. The requested data would be merged, filtered, sorted, printed on paper and delivered to the police commander. This process could take hours, sometimes days, depending on the complexity of the information request and the workload of CIS. But CIS could not meet data requests within seconds, so emergency personnel were despatched to locations with no knowledge of incidents at that location.

CIS and Cerebellum Software teamed to design an end-user application that would deliver information to police and crime personnel within seconds. The easy-to-use application, called Street Smarts, offers a Web-based graphical user interface (GUI) that allows users to perform database searches by selecting appropriate criteria. For example, a police officer using Street Smarts can access a variety of useful information, including all the burglaries reported for a

specific zone or address, crimes in the last 24 hours and the history of an address. With mobile data computers, Street Smarts can be used in the field. The solution is based on Cerebellum Software Inc.'s Cerebellum v1.2, which uses CORBA and the Java 2 platform to enable operating system and data independence. CORBA is used within the Cerebellum product to provide a communications layer between objects on different machines that need to share information. The use of CORBA enables Cerebellum and the City of Pittsburgh to integrate data from the various databases for presentation through one simple interface, without actually moving the data from the original sources.

CIS contracted Cerebellum Software, Inc. to implement the software and to train the IT staff to use the product and to develop new applications that could quickly and easily access data from disparate sources. The development of the Street Smarts application took approximately two months from concept through final presentation. Cerebellum makes fast application development possible because it eliminates the data access and integration work necessary to build new applications.

Because of the secure and personal nature of much of the crime and address information being accessed, Cerebellum's GUI allows CIS database administrators to manage user access. The Street Smarts application was also designed to limit the information that would be presented to end users.

3.2 Allied Signal – Vacation Time Tracking System

[7]AlliedSignal gathered requirements for a vacation system to track vacation time for employees across multiple business units and multiple sites with a deadline of two months. Specifically, AlliedSignal needed to monitor accruals, vacation days, partial days, exception days, historical information, and other vacation scheduling items. For tracking and monitoring purposes, external management reports needed to be printed and outside access to the vacation system was necessary.

AlliedSignal had to ensure the vacation system was secure, as it was storing and displaying Human Resources information. Employees needed access to their individual information in a secure manner, without the concern of unauthorized entry.

AlliedSignal had to implement the vacation system changes in a distributed environment while considering other factors such as a large user base and a very

short delivery schedule. There were multiple sites on wide area networks and almost one million transactions flowed into the database — 200,000 of which were complex high levels object calls. The total user population requiring access was approximately 250 users with most of the transactions happening on just three or four days of the month.

The final plan was a CORBA-based vacation system that was easy to use, scaleable, and performed well in a distributed architecture. Previously, the tracking, allocation, and scheduling tasks were not combined into a single system and information about unused vacation was not readily available. Now, when vacation information is entered into the vacation system, it is immediately available to those with access via web browsers. The vacation system is simple and most operations can be performed using only the mouse. It is accessible from any web browser and allows for widespread access and simplified configuration management since no part of the application is stored on the client machine.

Two previously built framework components are used to ensure security. First, the Security component addresses all authentication, authorization, and encryption. Second, a component called Trace is used to track requests for information throughout the vacation system and provide an audit trail. Using a Java-based applet calendar control, the user simply clicks their way through the vacation system to enter vacation days and make changes to the schedule. Vacation accruals are loaded through a batch process from the Payroll system. The primary benefits are ease of use for employees, simplified maintenance and support for the I/S staff, and lower cost for AlliedSignal. The vacation system utilizes a pre-built framework constructed in Java, C++, and Oracle.

The presentation layer of the vacation system uses Java applets. The business logic, which is stored on the server, was constructed in C++ and utilizes a pre-built framework for security, auditing, messaging, and communicating with the database. This framework is comprised of reused CORBA components which were built in 1995 and were awarded the "Best Application Utilizing Reusable Components" award at ObjectWorld ’97.

Use of CORBA has allowed AlliedSignal to leverage the benefits of the Java language (for portability), the benefits of the C++ language (for performance), and existing heterogeneous hardware (previous investments) in this vacation system. Additionally, the benefits of building reusable CORBA components (e.g. framework) has allowed for rapid development and reuse in other

applications to minimize costs and reduce cycle times.

The vacation system enjoys high employee satisfaction with the ability to control individual vacation time and better plan vacation usage. With the programmed complexities hidden with-in the vacation system, the end user is left with a simple point and click inter-face. Further, the vacation system was leveraged by other Business Units and is flexible enough to add additional employees dynamically.

3.3 Standard Chartered Bank – Customer Service Terminal

[6]Standard Chartered Bank in Hong Kong is using Orbix, a CORBA implementation by IONA Technologies, as the core infrastructure within its new application, the "Smart Customer Service Terminal", or SCST. SCST enables Customer Service Representatives (CSRs) to access a variety of service from a single desktop, helping them provide a timely and quality service to the bank's credit card customers. The application provides automation of one-stop services, for example, credit balance inquiry and bonus point inquiry. In addition, the SCST system automates general workflow, such as temporary credit limit increases, reporting of lost cards, and card repayment. Scripting is also automated to cater for product information and marketing promotion.

The SCST system takes a 3-tier client server approach, using Sybase 11 as a database engine running on RS/6000. Sun Microsystems' SunLink Gateway is used to connect Orbix Servers to the MVS host using LU 6.2. Web browsing capability is provided by Netscape Enterprise Server 3.0, while IE Active-X Control is used to integrate the Desktop and Web pages, in order to provide context sensitive scripting and a help facility. Bringing all these diverse technologies together as a single cohesive application is Orbix.

Orbix enables the connectivity required for a multi-tier architecture. Meanwhile Standard Chartered have also deployed OrbixNames in order to provide load balancing and facilitate location transparency between servers and clients. The implementation of OrbixManager for system management is also expected by the end of 1998.

The SCST is a large-scale CORBA project. Over 1,400 man-days have been spent developing the core of the SCST application, with an additional 400 to 600

to be expended during the rollout to the Asia Pacific region. At present the system supports 196 clients in the Asia Pacific region. CORBA was recognized early on as the integration technology of choice, enabling multi-platform interoperability and multi-tier architecture support. In addition, CORBA provides support for the end-to-end transaction-based workflow processing model used in the Smart Customer Service Terminal application. CORBA also facilitates the reuse of developed services, for example, the mainframe wrapper and the message macing server.

3.4 Harvard University – Educational Records System

[5]The Harvard Educational Records System was put into operation in the late 1970s. The system provided transaction processing for 11,000 undergraduate and graduate students in Harvard College and the Graduate School of Arts and Sciences. Like many systems of its time, it was efficient but complex to use, requiring highly trained operators. Furthermore, extracting analytical information for strategic decision making was extremely difficult. Increasingly, students, faculty, and administrators required direct access to critical information. To meet the needs of these constituents, a myriad of shadow systems had been developed.

The University administration recognized the need to replace this legacy system with a new information environment. The new system had to be Web-based to facilitate installation and support; it had to facilitate the rapid development of information access applications for students, faculty, and staff; and it had to integrate not only the existing legacy system, but multiple legacy data sources throughout the institution as well. Considering a variety of design alternatives, Harvard recognized that this mission-critical information system would need to meet as-yet-undefined needs over a long period of time. The ability to enhance and extend the system capabilities, therefore, was extremely important, as was the requirement for scalability to accommodate the extensions that would surely come.

Harvard elected to implement a three-tiered, distributed object solution. IT management selected Nevo Technologies to develop the system, considering the solution provider's extensive experience with Java and objects, as well as its proven implementation methodologies for introducing change into a large

operating organizations. Harvard's commitment to thin clients was critical to meeting two of the project's goals: ease of enhancement and reduced long-term support costs. With Java as the language of choice for both client and server development, and with Oracle already chosen for the primary database, the selection of the middleware component became the most important design decision. For this, Nevo selected VisiBroker from Inprise because of its dominance as the leading CORBA ORB, the overall maturity of the tool, its functional strength, and its complete integration with the Java environment.

Development of HERS-2, the second generation of the Harvard Educational Records System, got underway in July 1997. Beta release of the first modules occurred in December 1997, with full production release two months later in February 1998. The Faculty of Arts and Sciences Course Catalog, an 800-page document describing each of the 4,000 courses available to students, is the most visible output from the new system. The new system is dramatically easier to learn and use than its predecessor, HERS. Training time, formerly measured in months, now takes just hours. Most importantly, senior staff in the Office of the Registrar are freed to concentrate their efforts on larger professional responsibilities, rather than managing the nuts and bolts of operating a complex, cryptic computer system.

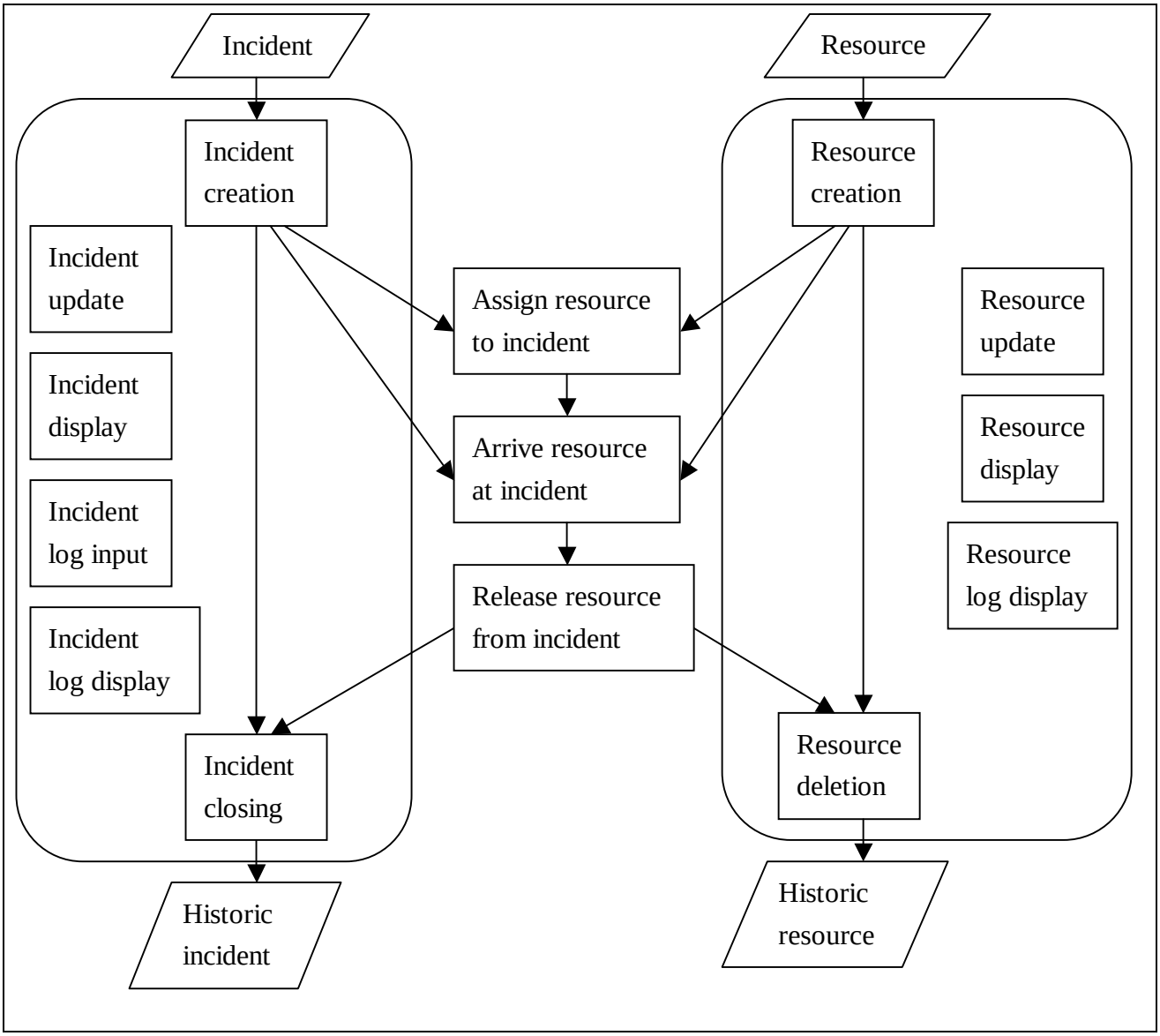
VisiBroker, a CORBA implementation from Inprise Corporation, enabled a system that maintains both the Oracle database and legacy UNIX files automatically, without any burden on either the application developers or end users. The benefit to Harvard is both a smooth transition to the new system from the old, and a complete integration of the legacy environment. Any functions that still await migration to HERS-2 can still be run by the legacy HERS-1.

The benefits to Harvard of a three-tiered architecture with thin clients, distributed business objects, and persistent data storage are substantial throughout the life cycle of the system. In development, the combination of Java and VisiBroker produce major gains in programmer productivity through both abstraction and reuse. In deployment, the use of thin clients eliminates the need to configure client machines and load drivers and software on each one. Furthermore, because the entry point to the applications is a browser, the environment is inherently familiar and comfortable, reducing training time and costs. In the maintenance area, Harvard benefits by eliminating the need to update client software and maintain and upgrade client configurations. Each time the user points his browser to the application location, the latest software is delivered.

The only requirement is that the browser be Java-enabled. Finally, and most importantly, the careful design of CORBA business objects provides a powerful, easy-to-use mechanism for the ongoing extension and enhancement of these applications in the future by clearly defining the interfaces, and, through the Visibroker ORB, providing universal access to them.

4 Computer Aided Despatch (CAD) System

Figure 3. Information flow of a computer aided despatch system.



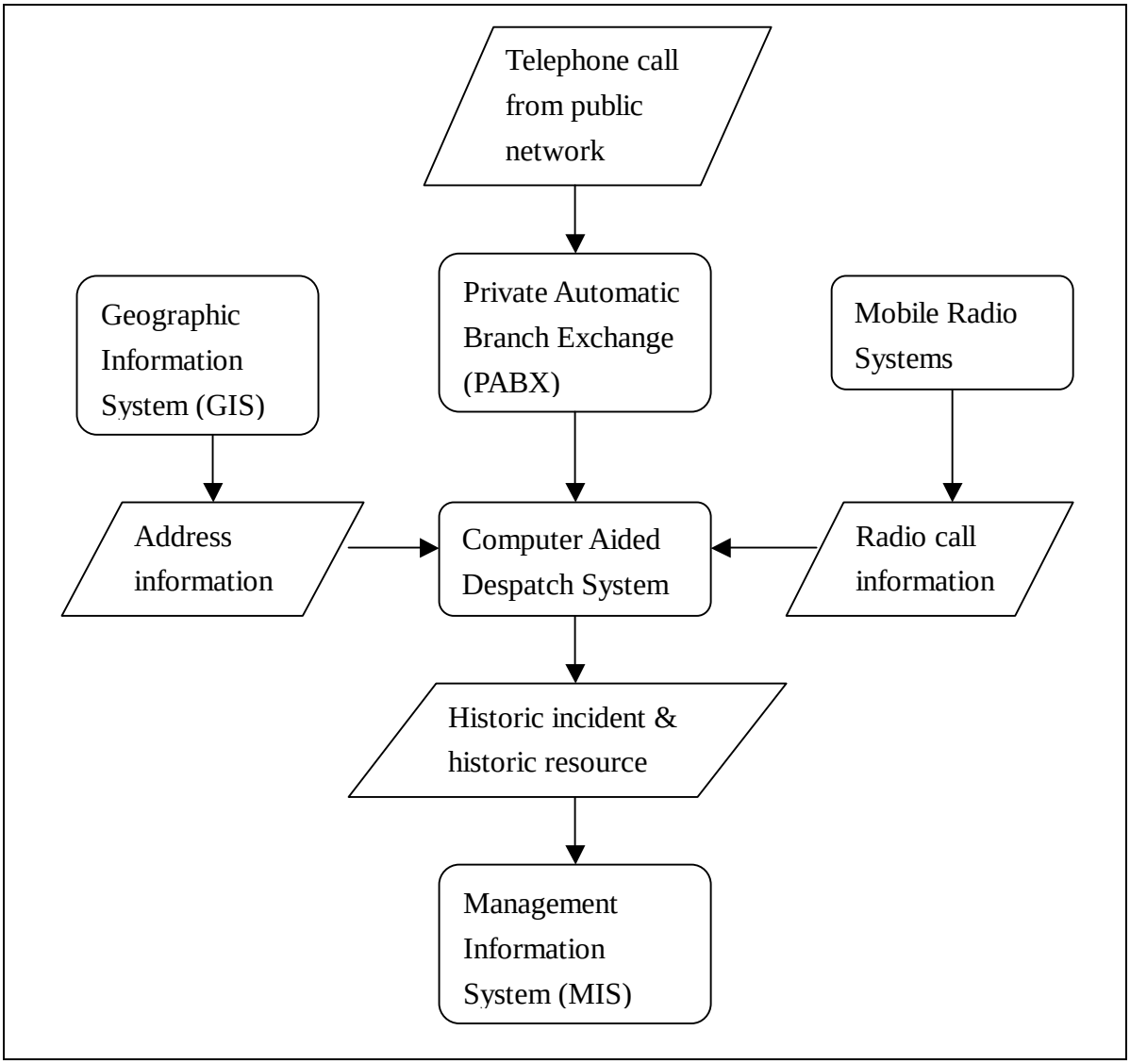
4.1 Information Flow

Figure 3 shows the information flow within a computer aided system. When an incident occurs, an operator enters it into the system using the *incident creation* function. After creation, the incident is either taken care of by resources, tracked using the *resource assignment, resource arrival or resource release* funcitons, or it is closed using *incident closing* function directly, possibly due to duplication or error. The r*esource assignment* function may be omitted, since a resource may discover an incident before the incident is reported by informants; in such case, assignment of resource is implied by the use of the *resource arrival* function. After all necessary immediate action is done, the incident can be transferred to other unit for follow up and the incident can be closed using the *incident closing* funciton. Notes that the incident is not physical purged, but it becomes an historical incident which is archived and used for reference in the future and for data analysis. During the lifetime of the incident, the *incident update* and *incident display* functions can be used to update and display the details of the incident respectively. Incident logs, which record the progress of the incident, can be input and displayed using the *incident log input* and *incident log display* functions.

When a resource begins its duty, an operator enters it into the system using the *resource creation* function. After that, the resource can be used with the *resource assignemnt*, *resource arrival* and *resource release* functions. When the duty time of the resource ends, the resource is removed from the system using the *resource deletion* function. Similar to handling of incidents, the resource is not physically purged, but it becomes an historical resource and is archived for future reference and data analysis. During the lifetime of the resource (in the particular duty), its details can be updated and displayed using the *resource update* and *resource display* functions. Resource logs, which record the interaction of the resource with incidents, can be displayed using the *resource log display* function.

4.2 Interaction with Other Systems

Figure 4. Interaction of a computer aided system with other systems.



A CAD system interact with many systems in order to obtain data for tracking incidents and resources, and to transfer out data for analysis. The private automatic branch exchange (PABX) switches and records telephone calls, and provides call data, like caller ID and caller address. Incidents are mostly reported through telephone calls. Operators also request assistance from other entities, like public utility companies and public transportation companies, by phone.

The mobile radio systems is the prime communication channel among resources and operators. The progress of incidents are mostly reported to the operator through the radio system. The system also generate call information, like caller

radio ID.

The geographic information system (GIS) provides accurate address information of incident location and resource location. These information enable the quick and efficient handling of incidents. The nearest resource can easily be located and assignment to an incident. The environment in the proximity of the incident scene can also be monitored, for example, availability of facilities and presence of danger can be seen and informed decisions can be made.

The historical data is transferred to the management information system (MIS) for analysis. Together with data from other data sources, online analysis processing (OLAP) and data mining can be done to extract valuable information.

5 System Requirements

5.1 Functional Requirements

There are two main entities in a CAD system: incident and resource. Incident is also called occurrence, problem report and something else depending on the target environment; it can be a taxi call, a electric fault report or a fire or robbery report. Resource can be a taxi, a technician, a team of fire fighters, or a team of police officers. The most important function of a CAD system is to despatch resource to incident, so that the incident is taken care of promptly.

Incident related functions include, but not limited to: incident creation, incident detail amendment, assignment of resource to incident, arrival of resource to incident, release of resource from incident, input of incident logs and closing of incident.

Resource related functions include, also not exhaustive: resource book-on, resource book-off, input of resource logs and resource detail amendment.

To enable prompt arrival of resource at incident location, some methods should be provided to indicate the closest resource to an incident. The best mean is the integration of Geographical Information System (GIS) with CAD system. GIS provide accurate location information of both resource and incident graphically, thus allowing precise matching of resource and incident. However, such integration is an extensive, involved exercise requiring man-years of work. In this project, a simple, yet effective, text-based gazetteer will be implemented to provide location information.

A CAD system essentially models the real world of incidents and resources. The user should be able to view the model in real time, that is, any change in the model should be reflected in the user interface immediately. This functionality requires the client to take the role of a server, to accept requests from server objects to update the incident and resource displays.

5.2 Performance Requirements

A CAD system is a kind of Online Transaction Processing (OLTP) system. The fundamental requirement is the quick and reliable processing of small unit of works, or transactions. To minimize the processing time taken by the system, both the computing time and the network transmission time should be minimized.

To minimized the computing time, efficient algorithms and protocols should be designed, and high performance database engine should be used. For example, when large amount of data is to be sent to the client, they should be sent one screenful at a time, triggered by paging command entered by the user. Server objects should be prestarted and any operations that are invariant of the requests, like connecting to the database engine, should be done when the server object is started, not when a request is coming. Care must be taken to write efficient SQL statements, and static SQL should be used where possible. Multithreading should be used in the user interface to reduce the perceived response time.

To minimized the network transmission time, interactions between remote objects should be kept at the minimum and interface definitions should be kept small, using the smallest possible date types.

In a production CAD system, the system response time is not simply the processing time taken by the software and hardware, but includes the handling time of the operator. To minimize the handling time, the user interface should be intuitive, and multiple means for giving command should be provided to suit users with different preferences. Drag-and-drop mechanism can be implemented to enable the user to manipulate incidents and resources as concrete objects, rather than abstract concepts. Both menus and command line should be provided to control the system.

5.3 System Management Requirements

A big problem in today's computer-enabled enterprises is system management.

No software is perfect, they need bugfixes and enhancements. In conventional 2-tier client-server systems, sophisticated yet complicated system management software is used to accomplish these tasks. Effort and time spent is substantial because of the fat clients running on heterogeneous platforms.

3-tier system together with Java applet provides a much better solution. By using the Java platform, only one version of the client software is needed, in spite of the potentially heterogeneous platforms. The client, implemented as a Java applet, is download to the client machine on its startup and on user request, so bugfixes and enhancements can be applied easily. The client in a 3-tier system is thin, so the downloading time is short. The middle tier of a 3-tier system, in which the business logic is implemented, runs on centrally managed servers, so their management is easy.

6 Outline Software Design Specification

6.1 Platform

Both the client and server objects will be written in Java and run in Java Virtual Machine (JVM), and the communication between them will be handled by CORBA ORB. The server objects will connect to a Relational Database Management System (RDBMS) via Java Database Connectivity (JDBC) for persistent storage. The client will be run as an Java applet inside Java-enabled Web browsers. The server will be run as standalone Java application, optionallly activated by the Object Activation Daemon.

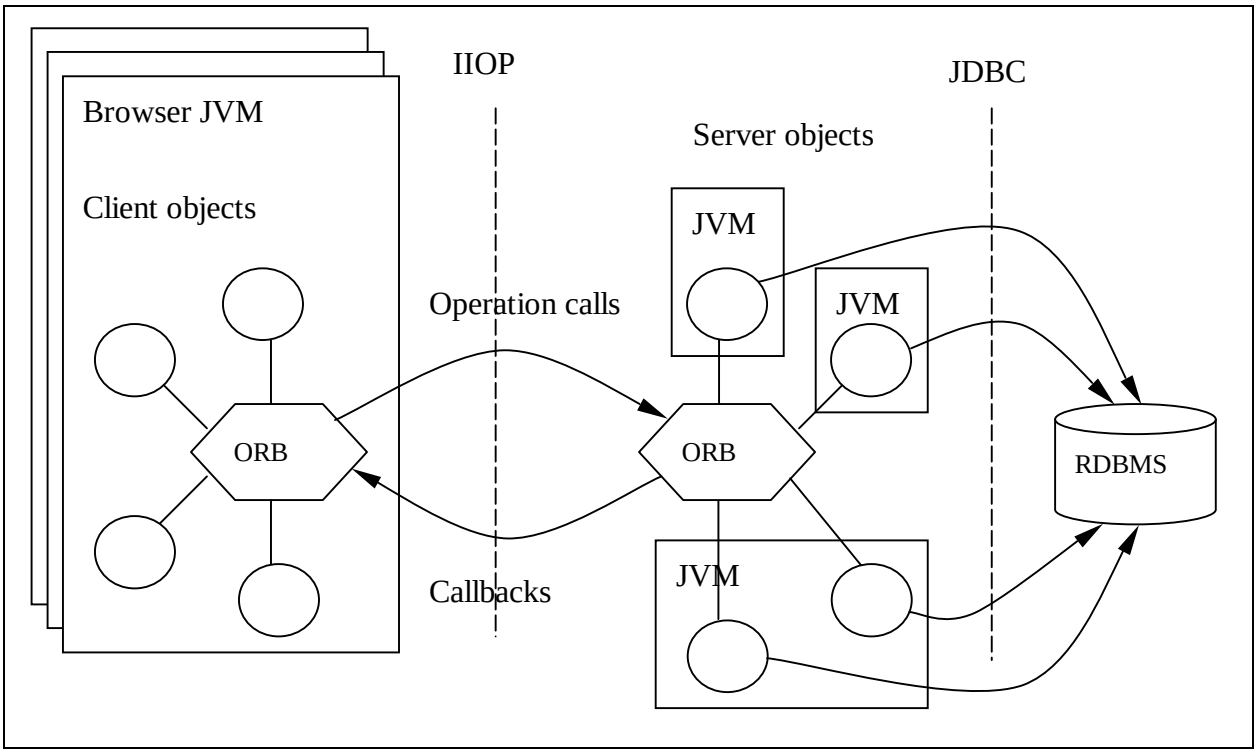
The specific software used in this project will be:

- Java Platform: Sun Microsystems Java Development Kit 1.2.2
- CORBA ORB: Inprise Visibroker for Java 3.4
- RDBMS: Oracle8*i*

Implementing the client and server using Java carries the "write once, run everywhere" Java advantage. Running the client as an applet eliminate the need to install ORB software in the client machine. Using CORBA as the middleware enable structuring the system in a 3-tier architecture. Running business logic and database operations in the middle tier – the server objects, the client is responsible only for the user interface and can be made thin, therefore

the downloading time of the applet is short and upgrading of database driver is done in the server, not the many clients.

6.2 System Architecture

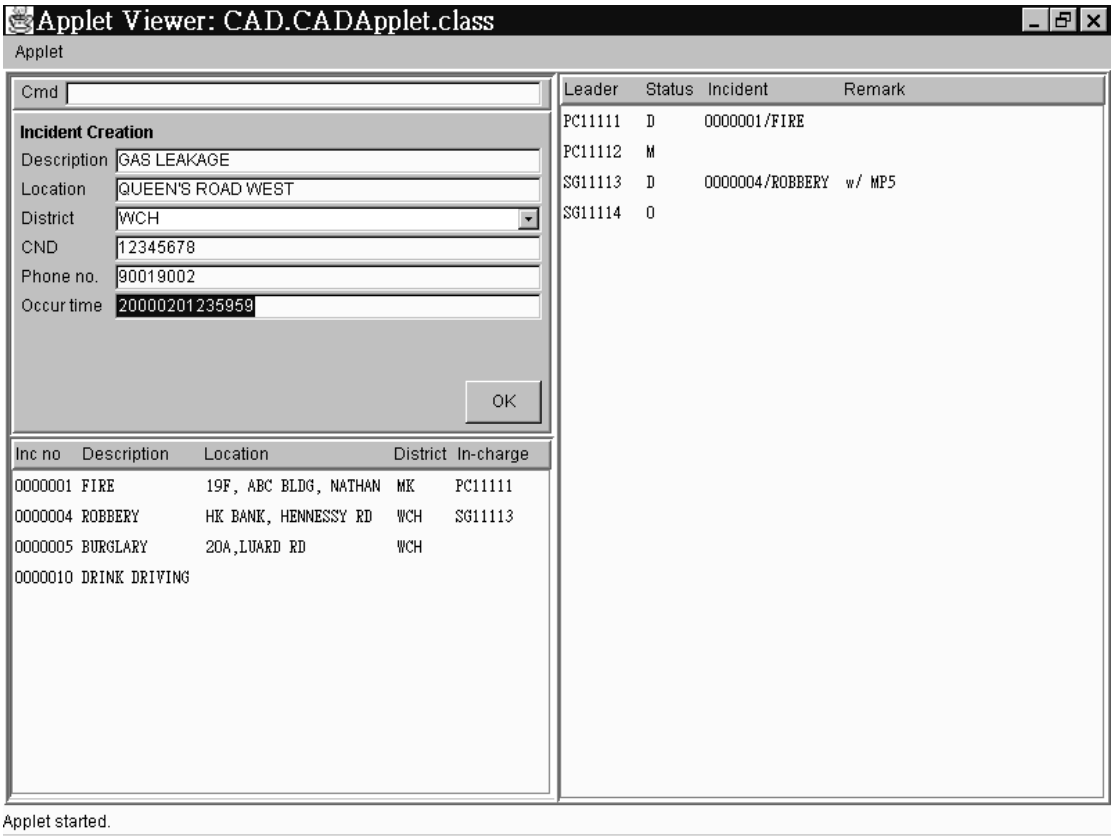


Client - The whole client runs inside the JVM of a Web browser, as an applet. The client consists of the ORB libraries and user-written objects, both of them are downloaded from a HTTP server as a result of the invocation of the Web page hosting the applet.

Server – The server also consists of the ORB libraries and user-written objects, but one or more server objects may run in a JVM, and several JVMs may be used to run the whole server.

RDBMS – The RDBMS has JDBC driver that the server objects can call to manipulate data using Structured Query Language (SQL).

6.3 User Interface



The user interface consists of three windows. One window is used for general interactions, like data input and inquiry data display. The other two windows are dynamic windows, that is, they are driven by the server as a result of data changes by the current user or other users. One of the dynamic windows is the incident queue, in which incidents are listed in reverse chronological order, that is, the latest incident is at the top of the queue. The other dynamic window is the resource list, in which resources under the user's control are displayed. The incidents and resources in the dynamic windows are not just lines of text, but behave like concrete entities. An incident can be dragged and dropped on an resource, and vice versa, to invoke operations between them. By double clicking on them, operations and inquiries can also be made.

7 Detailed Software Design Specification

7.1 General

The system is targeted for a fictitious emergency service. The services

provided include those provided by police, fire, ambulance and military departments. The area of the serviced region is in the order of hundreds of square kilometers, so that the whole region can be covered by tens of districts which is identified by district codes.

7.2 Database Schema Specification

Incident / Historic incident

Field	Data type	Remark
incident year	CHAR(4)	4-digit
incident number	CHAR(7)	1-9999999, restarted from one every year
description	VARCHAR	
location	VARCHAR	
district code	CHAR(4)	together with resources' district code, used to despatch resource to incident
reporting phone number	CHAR(10)	only local phone number are accepted, from Caller Number Display (CND)
contact phone number	CHAR(10)	only local phone number are accepted
create time	DATE	YYYYMMDDhhmmss
occur time	DATE	YYYYMMDDhhmmss
close time	DATE	YYYYMMDDhhmmss
status	CHAR(1)	A – awaiting action O – open (i.e. assigned or arrived resource) C – close
in-charge resource	CHAR(8)	leader resource identifier
close reason	CHAR(1)	R – resolved D – duplicated T – transferred E – error X - test
user	CHAR(8)	owner of the incident

Resource / Historic resource

Field	Data type	Remark
resource identifier	CHAR(8)	
type	CHAR(1)	P – Police F – Fire A – Ambulance M – military
radio identifier	CHAR(6)	resources carry walkie-talkie
leader resource identifier	CHAR(5)	"00000" if itself is a leader
district code	CHAR(4)	together with incidents' district code, used to despatch resource to incident
on-duty time	DATE	
off-duty time	DATE	

status	CHAR(1)	A – available D – deployed (assigned or arrived) M – meal O – other
user	CHAR(8)	owner of the resource

Operator

Operator corresponds to the person using the system. It is used for access control and as subject of accountability.

Field	Data type	Remark
operator	CHAR(5)	
password	CHAR(8)	stored in encrypted form
last accessing date	DATE	used to detect unauthorized access and house-keeping
last password change date	DATE	used to enforce password policies

User

User owns incidents and resources. Because the number of incidents and rescues can be very large, ownership is used to divide the workload, in order to guarantee the performance level. Because the underlying business services are provided for extended hours, or even on 24 x 7 basis, operators are on shift. Since incidents and working hours of resources may span across shifts, operator can not be used as owner of incidents and resources.

Field	Data type	Remark
user	CHAR(8)	
operator log-on count	NUMBER(3)	used to enforce at least one operator is logged-on

Incident log

Field	Data type	Remark
incident number	CHAR(7)	
log time	DATE	together with incident number, forms a unique key
log text	VARCHAR	

Resource log

Field	Data type	Remark
resource identifier	CHAR(8)	
log time	DATE	together with incident number, forms a unique key
log text	VARCHAR	

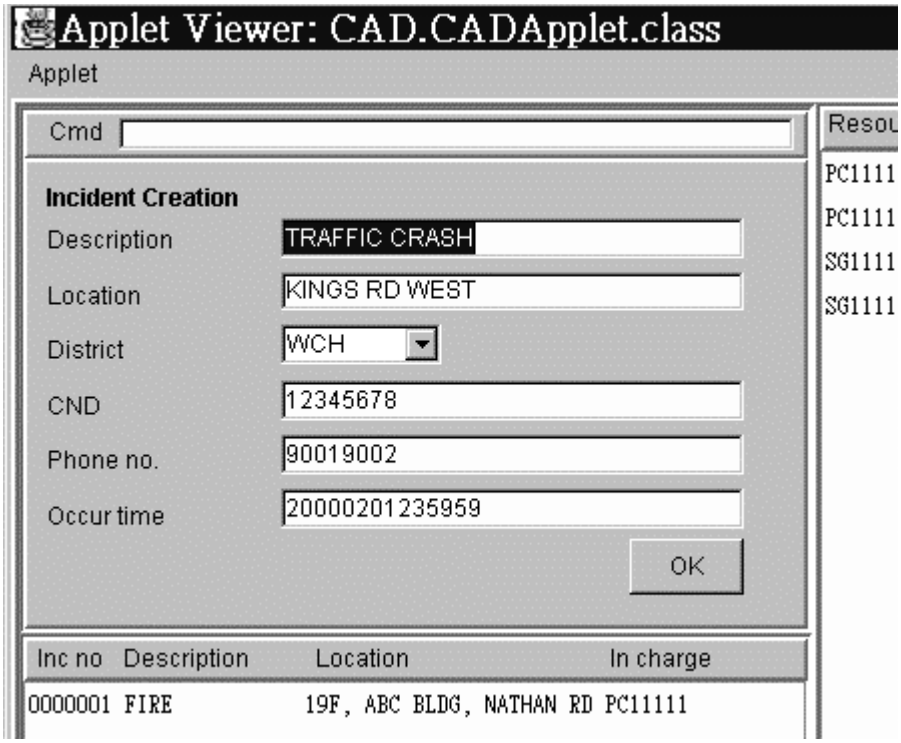
Incident-resource relation

Field	Data type	Remark
incident number	CHAR(7)	
resource identifier	CHAR(8)	
relation time	DATE	the latest relation is the current relation
relation type	CHAR(1)	A – assign V – arrive R – release

7.3 Function Specification

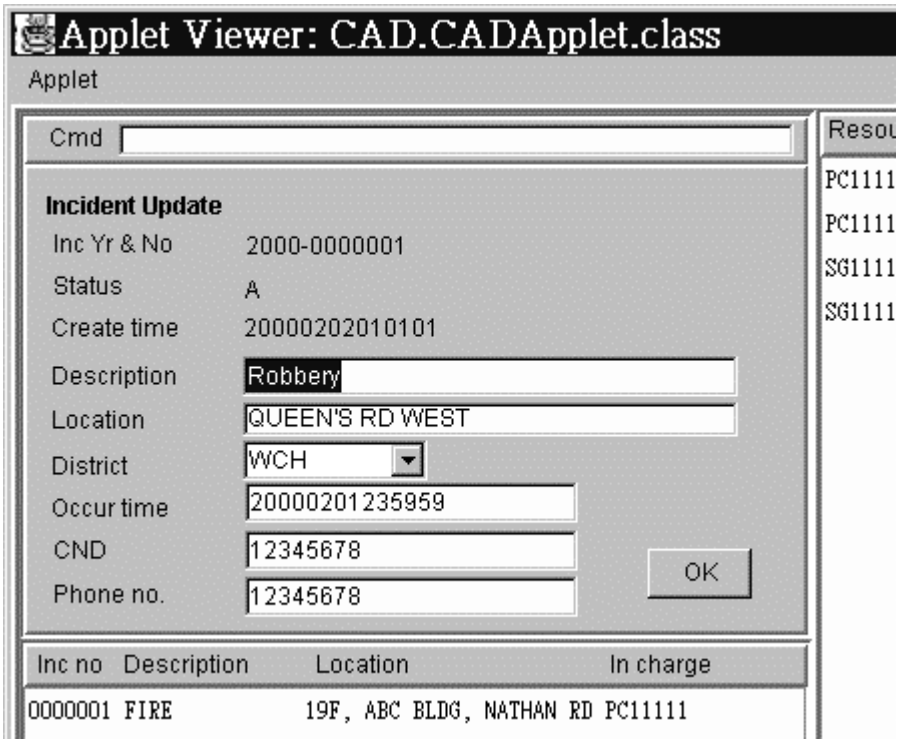
Incident creation

Operator action	Computer processing
Enter "IC" in the command line or invoke the menu *incident->create*	Display the *incident creation* panel
Enter information in the *incident creation* panel: • description • location • district code (via choice box) • reporting phone number • contact phone number • occur time (if it is a past incident) and click the OK button.	Insert an incident record into the database: • Set incident year to the current year • Set incident number maximum used + 1 • Set create time to the current time • If occur time is not entered, set occur time to the current time • Set status to A Insert the incident into all dynamic incident queues of the current user. Display *message* panel with message "Incident creation succeeded"



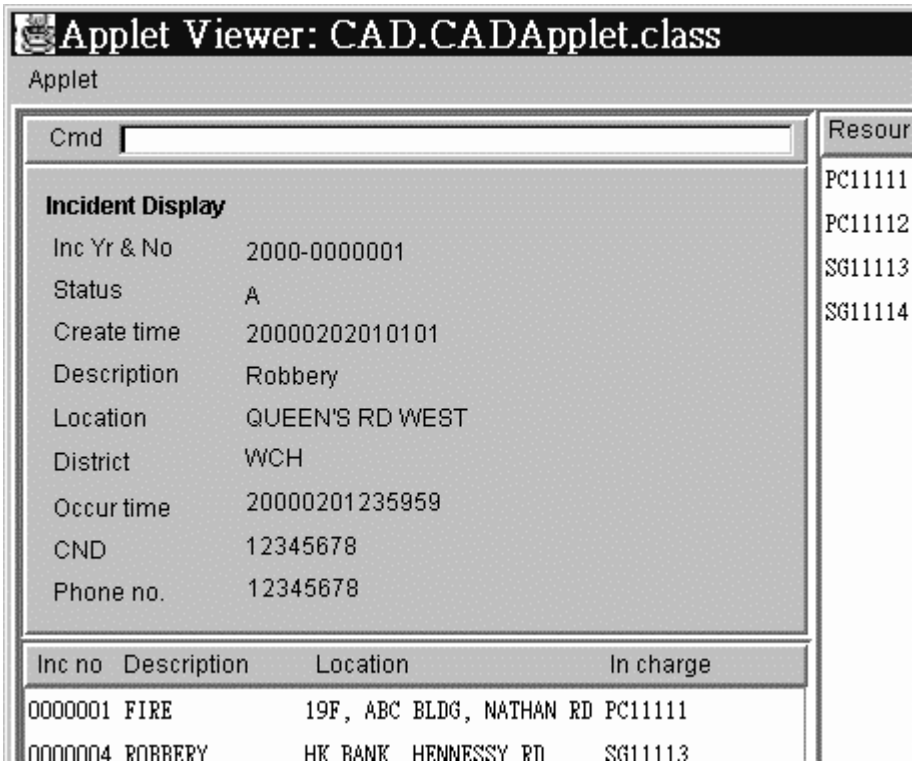
Incident update

Operator action	Computer processing
Enter "IU <incident number>" in the command line or invoke menu *incident->update* and enter incident number in the dialog box.	Display the *incident update* panel.
Update information in the *incident update* panel and click the OK button.	Update the incident record in the database Update the incident in all dynamic incident queues of the current user. Display *message* panel with message "Incident update succeeded".



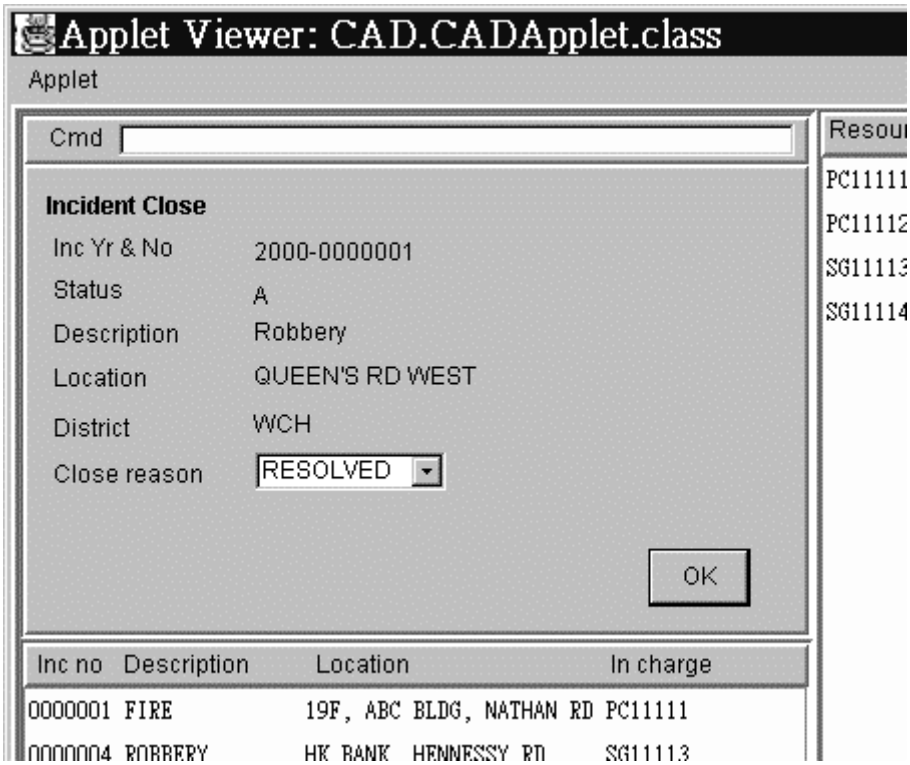
Incident display

Operator action	Computer processing
Enter "ID <incident number>" in the command line or invoke menu *incident->display* and enter incident number in the dialog box.	Display the *incident display* panel with information of target incident.



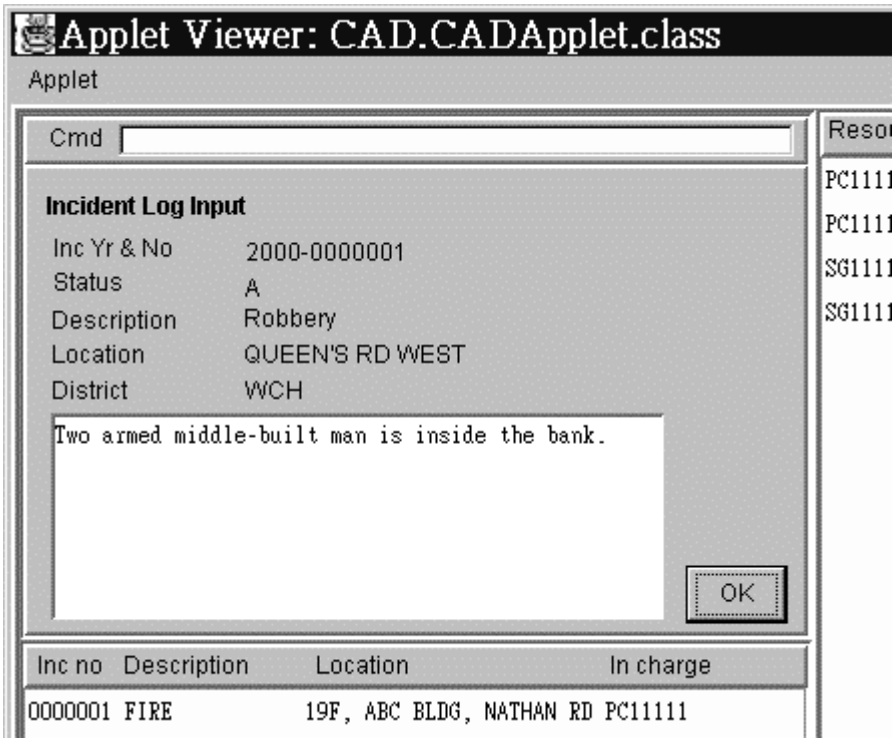
Incident closing

Operator action	Computer processing
Enter "ICL <incident number>" in the command line or invoke menu *incident->close* and enter incident number in the dialog box.	Display the *incident closing* panel.
Enter information in the *incident closing* panel and click the OK button.	Move the incident record to the historic incident table: • set state to "C" • set close time to current time Remove the incident from all dynamic incident queues of the current user. Display *message* panel with message "Incident closing succeeded".



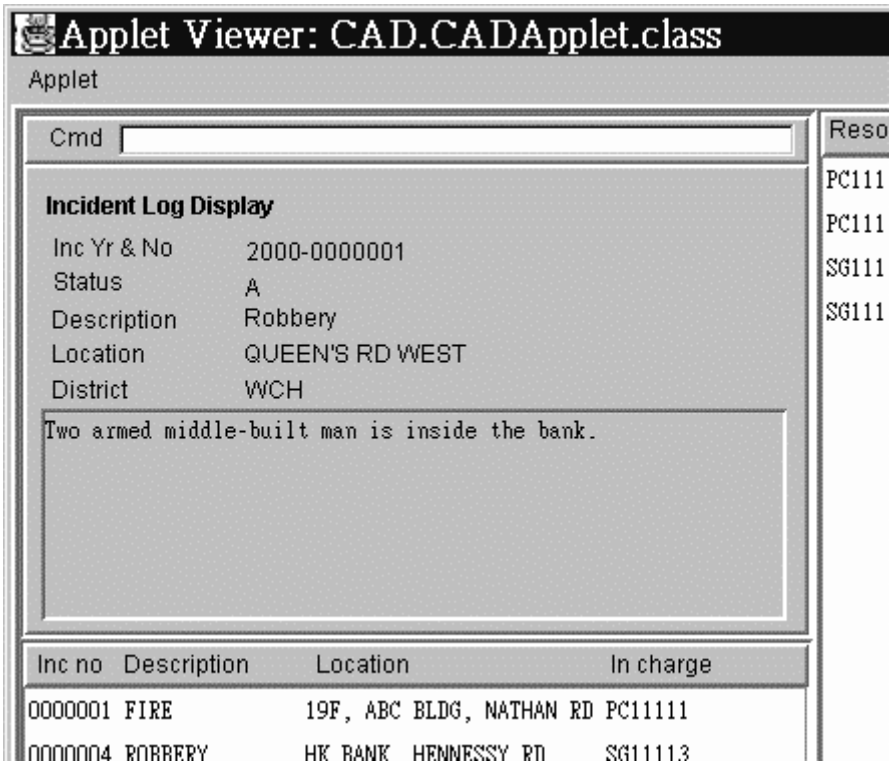
Incident log input

Operator action	Computer processing
Enter "IL <incident number>" in the command line or invoke menu *incident->add log* and enter incident number in the dialog box.	Display the *incident log input* panel with information of the target incident in short form.
Input logs in the *incident log input* panel and click the OK button.	Insert an incident log record into the database. Display the *message* panel with message "Incident log input succeeded".



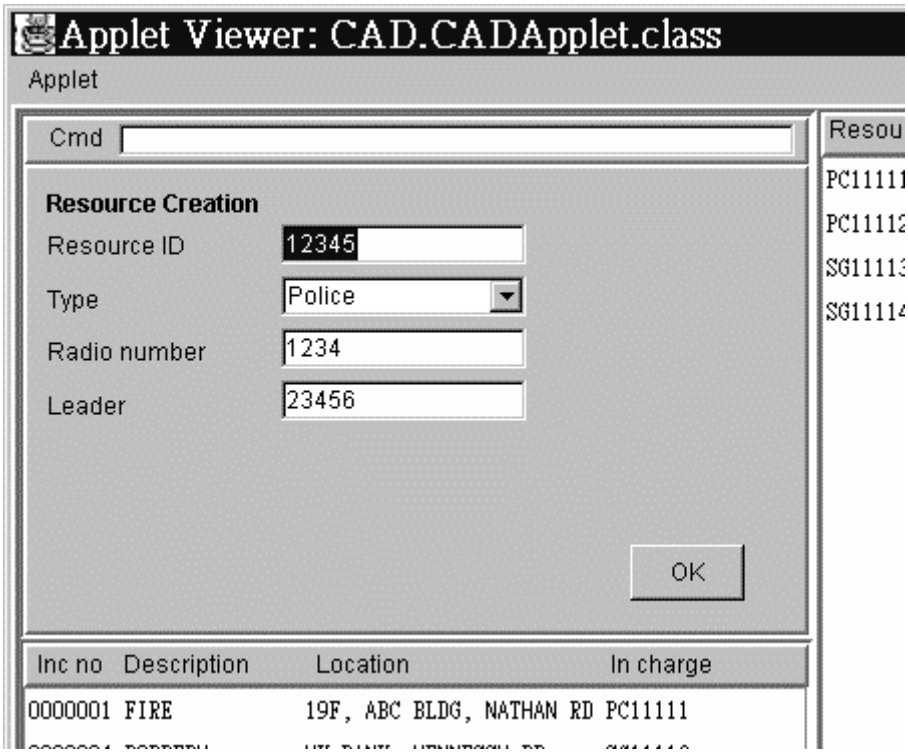
Incident log display

Operator action	Computer processing
Enter “ILD <incident number>” in the command line or invoke menu *incident->display log* and enter incident number in the dialog box.	Display the *incident log display* panel with information of the target incident in short form and all logs.



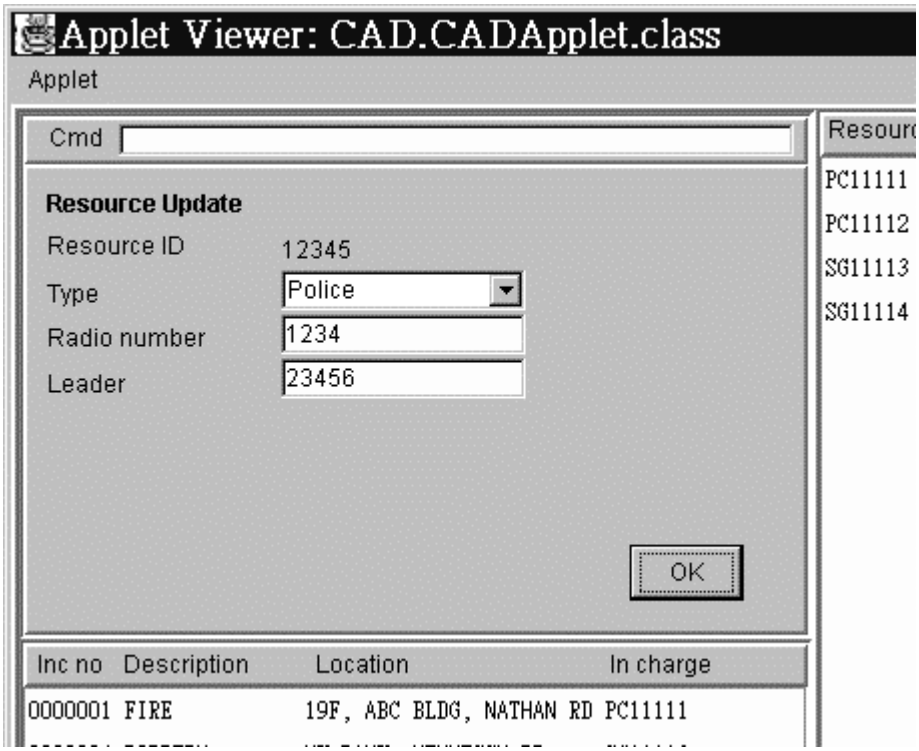
Resource creation

Operator action	Computer processing
Enter “RC” in the command line or invoke the menu resource->*create*	Display the *resource creation* panel
Enter information in the *incident creation* panel: • resource identifier • type (via choice box) • radio number • leader resource identifier and click the OK button.	Insert an incident record into the database: • Set on-duty time to the current time • Set status to A Insert the resource into all dynamic resource lists of the current user. Display *message* panel with message “Resource creation succeeded”



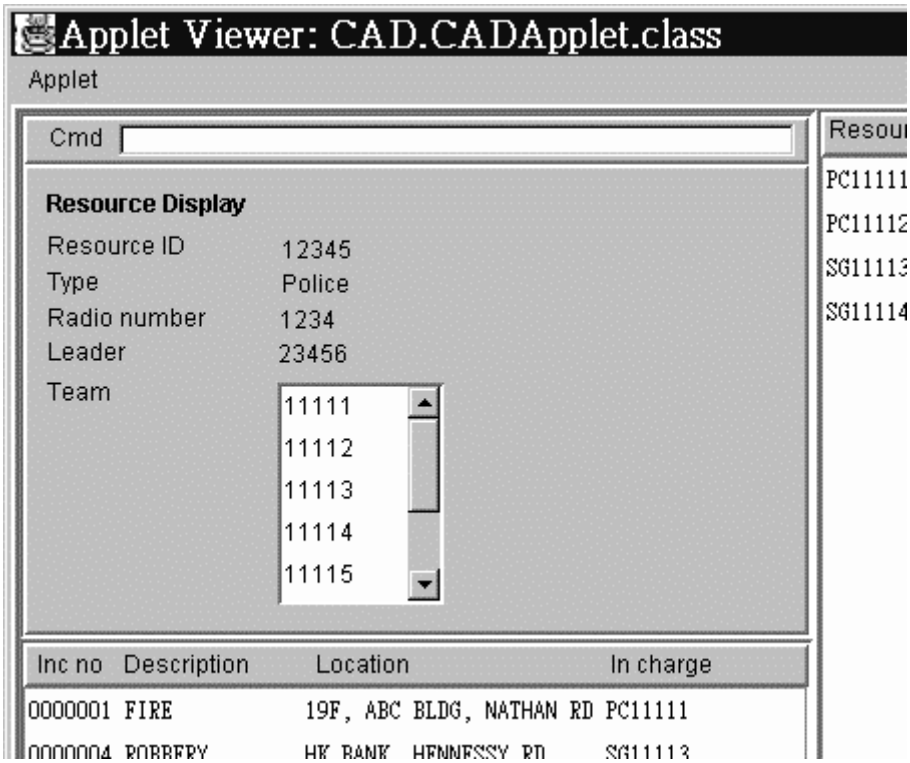
Resource update

Operator action	Computer processing
Enter “RU <resource identifier>” in the command line or invoke menu *resource->update* and enter resource identifier in the dialog box.	Display the *resource update* panel.
Update information in the *resource update* panel and click the OK button	Update the resource record in the database Update the incident in all dynamic resource lists of the current user.



Resource display

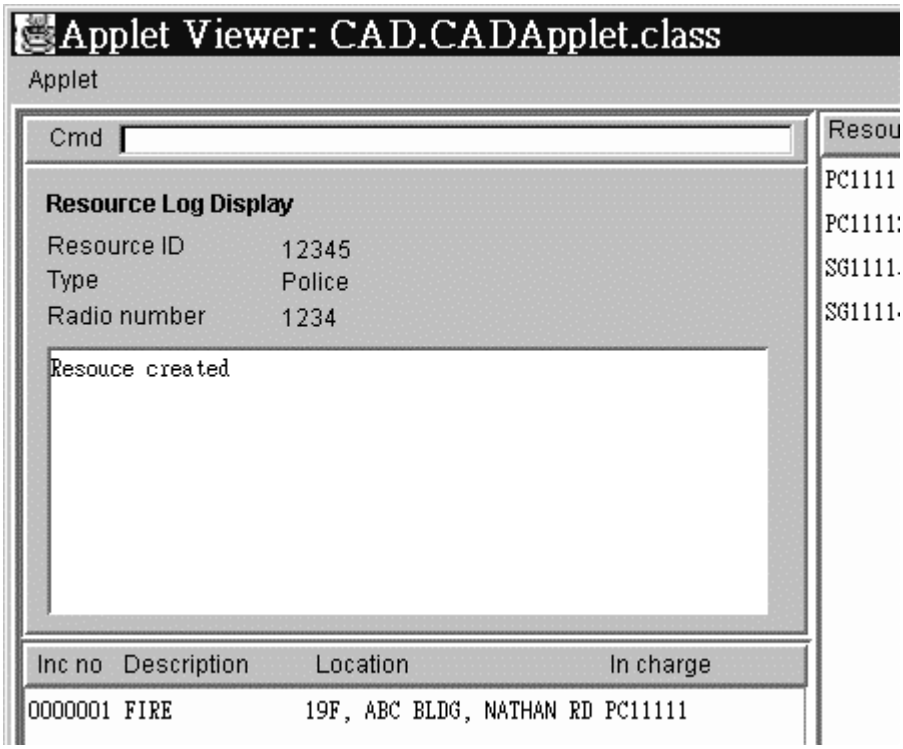
Operator action	Computer processing
Enter “RD <resource identifier>” in the command line or invoke menu *resource->display* and enter resource identifier in the dialog box.	Display the *resource display* panel with information of target resource.



Remark: either one of "Leader" or "Team" can have value, because a resource can be either leader or team member, but not both.

Resource log display

Operator action	Computer processing
Enter “RLD <resource identifier>” in the command line or invoke menu *resource->display log* and enter resource identifier in the dialog box.	Display the *resource log display* panel with logs of the target resource.



Resource delete

Operator action	Computer processing
Enter “RX <resource identifier>” in the command line or invoke menu *resource->delete* and enter resource identifier in the dialog box.	Validate that the resource status is available Move the resource record to historic resource table: • set off-duty time to current time

Assign resource to incident

Operator action	Computer processing
Enter “IRAS <incident number> <resource identifier>” in the command line or invoke menu *action->assign* and enter incident number and resource identifier in the dialog box.	Insert an incident-resource relation record of type “assign” into the database. Insert an incident log record into the database with text “Resource <resource identifier> is assigned.” Insert a resource log record into the database with text “Assigned to incident <incident number>.” Update the incident record in the database: • set in-charge to <resource identifier> Update the incident in all dynamic incident queues of the current user. Update the resource in all dynamic resource lists of the current user. Display the message panel with message “Resource assignment succeeded”.

Arrive resource at incident

Operator action	Computer processing
Enter “IRAR <incident number> <resource identifier>” in the command line or invoke menu *action->arrive* and enter incident number and resource identifier in the dialog box.	Insert an incident-resource relation record of type “arrive” into the database. Insert an incident log record into the database with text “Resource <resource identifier> arrives.” Insert a resource log record into the database with text “Arrive at incident <incident number>.”

	Update the incident record in the database: • set in-charge to <resource identifier> Update the incident in all dynamic incident queues of the current user. Update the resource in all dynamic resource lists of the current user. Display the message panel with message "Resource assignment succeeded".

Release resource from incident

Operator action	Computer processing
Enter "IRRL <resource identifier>" in the command line or invoke menu *action->release* and enter resource identifier in the dialog box.	Insert an incident-resource relation record of type "release" into the database. Insert an incident log record into the database with text "Resource <resource identifier> is released." Insert a resource log record into the database with text "Released from incident <incident number>." Update the incident in all dynamic incident queues of the current user. Update the resource in all dynamic resource lists of the current user. Display the message panel with message "Resource release succeeded".

7.4 Interface Definitions

```
module CAD {

    // Interface to be implemented as server side object.
    // providing incident related functions.

    exception DatabaseError { string message; };
    exception OperationFailed { string message; };

    interface Incident {

        void create(in string description,
                    in string location,
                    in string district,
                    in string CND,
                    in string phoneNum,
                    in string occurTime,
                    in string user)
                    raises( DatabaseError );
```

```
void update(in string incYear,
            in string incNumber,
            in string description,
            in string location,
            in string district,
            in string CND,
            in string phoneNum,
            in string occurTime,
            in string user)
            raises( DatabaseError,
                OperationFailed );

void display(in string incNumber,
            out string incYear,
            out string description,
            out string location,
            out string district,
            out string CND,
```

```
            out string phoneNum,
            out string createTime,
            out string occurTime,
            out string closeTime)
        raises( DatabaseError,
            OperationFailed );

    void close(in string incNumber,
            in string closeReason)
            in string user)
        raises( DatabaseError,
            OperationFailed );

    void inputLog(in string incNumber,
            in string logText,
            in string user)
        raises( DatabaseError,
            OperationFailed );
```

```
void displayLog(in string incNumber,
                in string lastReadTime,
                out string logText)
                raises( DatabaseError,
                    OperationFailed );

void assignResource(in string incNumber,
                in string resID,
                in string user)
                raises( DatabaseError,
                    OperationFailed );

void arriveResource(in string incNumber,
                in string resID,
                in string user)
                raises( DatabaseError,
                    OperationFailed );

void releaseResource(in string resID,
```

```
                in string user)
                raises( DatabaseError
                    OperationFailed );
};

// Interface to be implemented as server side object
// providing resource related functions.

interface Resource {

    void create(in string resID
        in char type
        in string radioID
        in string leaderResID
        in string user)
        raises( DatabaseError
            OperationFailed );

    void update(in string resID
```

```
        in char type,
        in string radioID,
        in string leaderResID,
        in string user)
        raises( DatabaseError,
            OperationFailed );

    void display(in string resID,
        out char type,
        out string radioID,
        out string leaderResID,
        out string onDutyTime,
        out string offDutyTime,
        out char status)
        raises( DatabaseError,
            OperationFailed );

    void delete(in string resID,
        in string user)
```

```
            raises( DatabaseError.
            OperationFailed ).

        void displayLog(in string resID.
            in string lastReadTime.
            out string logText)
            raises( DatabaseError.
                OperationFailed ).
};

// Interface to be implemented as client side object.
// for the add. update and delete of entries in the
// dynamic incident queue and resource list.

interface IncQueue {
        void addInc(in string incNum.
            in string description.
            in string location.
```

```
            in string district,
            in string inChargeRes,
            in char status)
            raises( OperationFailed );

        void updateInc(in string incNum,
            in string description,
            in string location,
            in string district,
            in string inChargeRes,
            in char status)
            raises( OperationFailed );

        void deleteInc(in string incNum)
            raises( OperationFailed );
};

// Interface to be implemented as client side object.
// for the add, update and delete of entries in the
```

```
// dynamic resource list.


interface ResList {

    void addRes(in string resID.

        in char type.

        in char status.

        in string inChargeInc)

        raises( OperationFailed ).


    void updateRes(in string resID.

        in char type.

        in char status.

        in string inChargeInc)

        raises( OperationFailed ).


    void deleteRes(in string resID)

        raises( OperationFailed ).

};
```

```
// Interface to be implemented as server side object.
// as a central switching point of dynamic windows
// update requests.

interface DynaWinHandler {

    void register(in IncQue incQue.
        in string user)
        raises( DatabaseError.
            OperationFailed ).

    void register(in ResList resList.
        in string user)
        raises( DatabaseError.
            OperationFailed ).

    void addInc(in string user.
            in string incNum.
```

```
        in string description,
        in string location,
        in string district,
        in string inChargeRes,
        in char status)
        raises( DatabaseError,
            OperationFailed );

void updateInc(in string user,
        in string description,
        in string location,
        in string district,
        in string inChargeRes,
        in char status)
        raises( DatabaseError,
            OperationFailed );

void deleteInc(in string user,
        in string incNum)
```

```
        raises( DatabaseError,
            OperationFailed );

void addRes(in string user,
        in string resID,
        in char type,
        in char status,
        in string inChargeInc)
        raises( DatabaseError,
            OperationFailed );

void updateRes(in string user,
        in string resID,
        in char type,
        in char status,
        in string inChargeInc)
        raises( DatabaseError,
            OperationFailed );
```

```
        void deleteRes(in string user,
                       in string resID)
                       raises( DatabaseError,
                           OperationFailed );
    };
};
```

7.5 Server-side Classes

CORBA Object implementations

IncidentImpl – implements the Incident interface

ResourceImpl – implements the Resource interface

DynaWinHandlerImpl – implements the DynaWinHandler (Dynamic Windows Handler) interface

Object Servers

IncidentServer – instantiates and runs IncidentImpl

ResourceServer – instantiates and runs ResourceImpl

DynaWinHandlerServer – instantiates and runs DynaWinHandlerImpl

7.6 Client-side Classes

CORBA Object implementations

IncQueImpl – implements IncQue interface

ResListImpl – implements ResList interface

Applet

CADApplet – instantiates and runs IncQueImpl, ResListImpl, and handles user actions.

PnlIncCreate – extends JPanel and handles the incident creation function.

PnlIncUpdate – extends JPanel and handles the incident update function.

PnlIncDisplay – extends JPanel and handles the incident display function.

PnlIncClose – extends JPanel and handles the incident closing function.

PnlIncLogInput – extends JPanel and handles the incident log input function.

PnlIncLogDisplay – extends JPanel and handles the incident log display function.

PnlResCreate – extends JPanel and handles the resource creation function.

PnlResUpdate – extends JPanel and handles the resource update function.

PnlResLogDisplay – extends JPanel and handles the resource log display function.

PnlResDisplay – extends JPanel and handles the resource display function.

8 Project Schedule

From	To	Task	Remark
mid January	mid February	Prepare project proposal	
mid February	mid March	Prepare detailed software design specification	Interfaces in IDL, function of classes and their methods.
mid March	mid April	Build user interface	
mid April	mid May	Build server classes	
mid May	mid June	Build database connectivity	
mid June	mid July	Perform system integration and testing	
mid July	mid August	Prepare documentation	

9 References

1. Andreas Vogel and Keith Dubby, *Java Programming With CORBA*, 2nd edition (Wiley, 1998)
2. Robert Orfali and Dan Harkey, *Client/Server Programming with Java and CORBA*, 2nd edition (Wiley, 1998)
3. Douglas Bell and Mike Parr, *Java for Students*, 2nd edition (Prentice Hall Europe,

1999)
4. G. Coulouris, J. Dollimore and T. Kindberg, *Distributed Systems: Concepts and Design*, 2nd edition (Addison-Wesley, 1994)
5. Inprise Corporation, *VisiBroker Case Study* (2000) <http://www.inprise.com/visibroker/cases>
6. IONA Technologies, *IONA Customers* (2000) <http://www.iona.com/info/aboutus/customers/index.html>
7. Object Management Group, *CORBA Success Stories* (2000) <http://www.corba.org/>
8. Java Development Kit 1.2.2, Sun Microsystems, 1999
9. Thomas J. Mowbray and Raphael C. Malveau, *CORBA Design Patterns* (John Wiley & Sons, Inc., 1997)
10. Thomas J. Mowbray and William A. Ruh, *Inside CORBA: Distributed Object Standards and Applications* (Addison Wesley Longman, Inc., 1997)