OAM Software Design Document
CS 446 – Spring 2001

GROUP #1
Troy Gonsalves 97083748
tgonsalves@student.math.uwaterloo.ca

Chris Mennie 97024327
camennie@student.math.uwaterloo.ca

David Tapuska 97084449
dftapusk@student.math.uwaterloo.ca

June 25, 2001

1.0 Abstract

The purpose of this document is to guide a small team of programmers to efficiently implement an operational SX4 OAM system. This system is used by operators for the management of a small telephone exchange system. This system will be used to manage customer records, billing history, billing payments, and control the underlying hardware.

This document will provide sufficient design and implementation details to allow novice programmers complete the project.

All interaction between each component module is described, and details given of the interfaces used. Between the UI Clients and the OAM Server a TCP/IP connection is used, over which XML messages are passed. A MySQL database is used to store the system's information. TCP/IP is used to communicate between the OAM Server the database.

As specified in the OAM Software Interface specs, communication between the OAM Server and the CUs will use message queues.

A prototype UI Client is provided. The UI Client will be implemented using Tcl/Tk.

For the convenience of project leaders or management, cost and time estimates are provided.

2.0 Introduction

2.1 Purpose

This document is intended to provide a detailed description of the SX4 OAM software system, for use in implementation. It is intended that this document be used by both developers and project leaders/management.

For developers, this document will serve as a template for implementation. Detailed descriptions of the main classes and methods found within the system modules are given. As well, an appropriate coding standard is provided.

For project leaders or managers, this document provides information about development and maintenance costs, and a task schedule.

2.2 Scope

As described in the OAM Architecture Document, each process of the OAM consists of a number of modules. The functionality of each of these modules is summarized in a fashion similar to a CRC card, followed by descriptions of the main routines and classes found within. Pseudo-code examples of complex functions are provided. As well, examples of inter process communication that occurs due to function calls are provided.

Externally visible interfaces are presented, so that developers can quickly reference the details when they need access to a component. UI descriptions, as well as external communications interfaces are provided.

An Integration Task Plan is proposed to allow concurrent development. Following the plan given will yield major milestones at which integration and integration testing can be performed.

Cost estimates are given, in terms of lines of code and time to implement, for each module. It is expected that under normal circumstances, the time to develop would be 9 months, at a cost of $105,600, with an annual maintenance cost of $25,600. A task schedule is provided in the form of a Gantt chart. A period of time between June 30th and July 2nd has been recognized as a likely period of vacation for all developers on the project.

The coding standards will adopt a form of Hungarian notation as a naming convention, and follow the GNU Coding Standards for formatting the source code (with slight modifications). Other customized standards are also described.

2.3 Document Conventions

This document makes use of many tables to convey method and process information. Diagrams are provided to further augment understanding. The method tables will follow the following format:

Declaration
Provides either a C/C++ member function declaration:
`returnType functionName(arg1,arg2,...)`
or a Tcl/Tk procedure declaration:
`proc procedureName {arg1 arg2 ...}`

Purpose What this method does
Arguments (optional) Description of input parameters
Return Value (optional) Description of any explicitly returned values (values returned through input arguments are described in the Arguments section)
Implementation Notes/Pseudocode (optional) Specifies any additional notes that are important to the implementation and/or provides a pseudocode.

The table for processes is similar except that is specifies ***Usage*** instead of specifying method ***Declaration***. For usage signatures, <argument> represents a required argument and [argument] represents an optional argument.

When specifying XML messages sent or received, the type of message is usually encompassed in < > brackets. This denotes that that type of message is expected in the XML, along with the structure of that node that is specified in the DTD. The method may specify that <TestEquipmentRequest> is an argument, but that actually corresponds the message:

```
<Message>
  <TestEquipmentRequest>
    <EX>2</EX>
    <Shelf>2</Shelf>
    <Slot>3</Slot>
  </TestEquipmentRequest>
</Message>
```

3.0 Design Overview

3.1 Module Dependency Diagram

The following diagram displays module dependencies. Dashed lines indicate messaging dependencies. Messaging dependencies are those that the interface between these components is determined to be an external interface is well defined. And hence there is not a direct dependency.

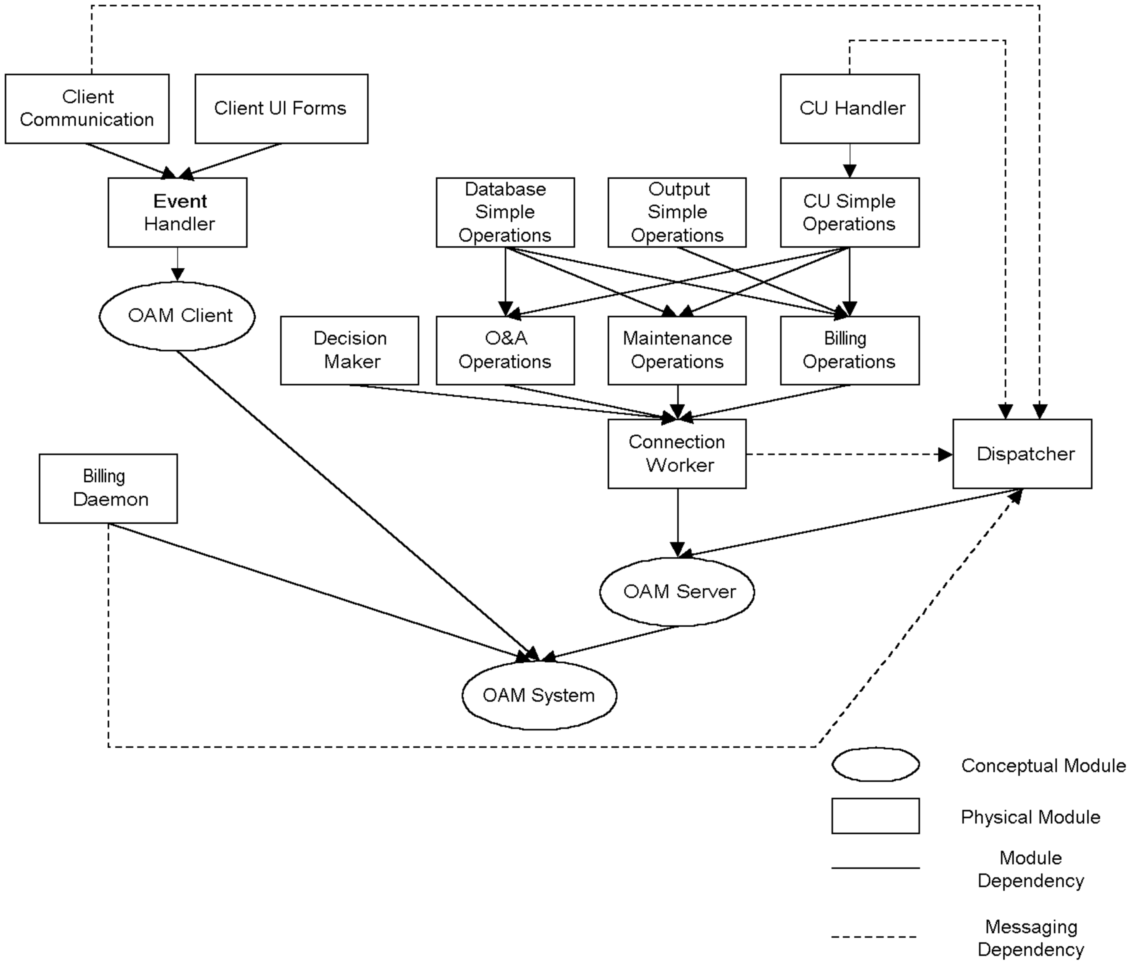


Figure 1 Module Dependencies

3.2 Design Decisions and Justification

The OAM System follows a Client Server Design Pattern. This is typical of a system that maintains and manages central data that can be accessed from many points across the network. Some of the design considerations and justification of choices of how the system is partitioned are presented in this section. It includes Class-Responsibility-Collaborator (CRC) cards for each major module. The Detailed

Design section will further expand on each of the modules outlined by expressing externally visible methods to other components.

3.2.1 Internationalization

A consideration that was present in all of our design decisions was to allow Internationalization of the end visible product. Strong typing of messages and structures by enumerated types permits this. For example, our error return results from the Server to Client are well define integers and not error messages. The UI can then look at its locale table for resources to get the language specific message corresponding to the error. It would be difficult to implement an internationalized version of a product in the short amount of time that is projected; we feel that the product produced will require some work for it to be internationalized but it should not require any architectural or major design changes.

3.2.2 Messaging

The Messaging System between major components is designed to be one that uses well formatted XML. There are numerous benefits to this, including validation, extensibility and abstraction. The XML messages sent between components can be readily verified for the correct structure using a Document Type Definition (DTD). The DTD is specified in the External Interfaces Section.

Using XML allows us to abstract the server from the client completely, in that the server can be written independent of the client being written. The server need not know about the OS or language that the client is running. XML also allows us to support multiple languages, and not have to change the protocol in which messages are sent out. In other words, we can write multi-byte character sets into the XML and dynamically determine what character set to use on the server end.

Another benefit of using XML is that it can be easily captured, and regression test can be done on individual components.

3.2.3 OAM Server

The OAM Server is partitioned into two components, the Dispatcher and the Connection Worker. The partitioning occurs based on functionality. The Connection Worker is responsible for servicing a client’s request. The Dispatcher is responsible for administration of Connection Workers, such as allocating and notifying. The design pattern that this separation of functionality corresponds to is commonly referred to as the Administrator-Worker pattern. It allows many clients to access the various services, while maintaining a high level of concurrency.

The Connection Worker is partitioned using a layered Design Pattern. It has three main layers, the Decision Maker layer, the Complex Operations layer and the Simple Operations layer. The divisions of the layers occur based on complexity of each layer.

3.2.3.1 Dispatcher & Connection Worker

Dispatcher	
Collaborator	**Responsibility**
• Connection Worker • Billing Daemon • OAM Client • CU Handler	The Dispatcher is the central point for the distribution and synchronization of work; it is an administrator for client processes. It delegates requests from the OAM Client and CU Handler. A client begins by establishing a connection to the Dispatcher. The Dispatcher then allocates a Connection Worker to handle the client's request. The Dispatcher can also broadcast messages to all existing Connection Workers to notify them of system errors.

Connection Worker	
Collaborator	**Responsibility**
• Dispatcher • Billing Daemon • OAM Client • CU Handler • Database • Print Server	The Connection Worker handles all communication between the OAM Server and a client. A Connection Worker has three sub-components; Decision Maker, Complex Operations and Simple Operations. All client requests are processed through the decision maker and passed down to complex and simple operations that are performed on low-level operations on the resources in the system.

3.2.3.2 Decision Maker

Decision Maker	
Collaborator	**Responsibility**
• Connection Worker • Complex Operations • Dispatcher • OAM Client • CU Handler	The Decision Maker handles all communication between the OAM Server and a client. Once created a Connection Worker's Decision Maker is the only component that can communicate with the client.

3.2.3.3 Complex Ops layer

The Complex Operations is subdivided into three modules; Operations & Administration, Maintenance, and Billing. These three modules compose the core functionality of the system. The division of the Complex Operations occurs based on similar functionality.

Operations & Administration

Collaborator	Responsibility
• Decision Maker • Simple Operations	This component handles Operations & Administration type requests. The component communicates with the three Simple Operations Layer. An example of such a request is Add Customer.

Maintenance	
Collaborator	**Responsibility**
• Decision Maker • Simple Operations	This component handles Maintenance type requests. The component communicates with the three Simple Operations Layer. An example of such a request is Reset a Line Card in the CU.

Billing	
Collaborator	**Responsibility**
• Decision Maker • Simple Operations	This component handles Billing type requests. The component communicates with the three Simple Operations Layer. An example of such a request is Print the bill for a customer.

3.2.3.4 Simple Ops layer

The bottom Simple Operations layer is divided into three components, Database, Output and CU. These three divisions occur based on functional components that each communicates with. Components on this layer are not allowed to communicate amongst each other. The original design of the Database Simple Operations was to follow a Create, Restore, Update, Delete (CRUD) design pattern. However, it was found that CRUD produced an inefficient interface for doing operations. For example, if we needed to do something on all subscriptions, we must fetch all the subscriptions then execute an update on each one. For large databases this would create many database execution messages to be performed, whereas it could be done in one single database message. Therefore, our Database Simple Operations interface has a CRUD look and feel for some of the operations, but we have extended its functionality for optimization purposes.

Database Simple Operations	
Collaborator	**Responsibility**
• Complex Operations • Database	Provides a set of basic methods for interfacing with the Database. These methods are similar to system calls in an operating system.

CU Simple Operations

Collaborator	Responsibility
• Complex Operations • CU Handler Operations	The CU Simple Operations layer implements basic atomic operations and provides an abstract interface to the CU.

Output Simple Operations	
Collaborator	**Responsibility**
• Complex Operations • Print Server	The Output Simple Operations layer implements basic atomic operations and provides an abstract interface to the PDF bill generation, emailing of bills and physical outputting of bills.

3.2.4 OAM Client

The OAM Client uses an adaptation of the Model, View and Controller design pattern. The Event Handler represents the Controller, the Communications Layer represents the Model, and the UI Forms represent the View. The hierarchical and navigational feel of the GUI to be design differs from the initial specification of it in SRS. These changes along with screen captures and functionality are presented in Appendix B.

Communication Layer	
Collaborator	**Responsibility**
• OAM Server • Event Handler	The Communication module acts as an interface between the Event Handler and the Server. It is responsible for transforming the internal data representation into the externally visible data representation of XML. It handles requests received from both the Server and Event Handler and deals with them appropriately.

UI Forms	
Collaborator	**Responsibility**
• Operator • Event Handler	The User Interface Forms module acts as an interface between the OAM Client's Event Handler and the Operator Terminal. This involves formatting the display that is output by the Operator Terminal and altering this display when events and data are received from the Event Handler. The module must also pass events and data from the Operator Terminal to the Event Handler.

Event Handler

Collaborator	**Responsibility**
• Dispatcher • Connection Worker • UI Forms	The Event Handler defines methods for all operations that are available to the OAM Client. These methods are used to decide which events are sent to the OAM Server in response to input events from a User Interface Forms. For example, pressing a button to add a customer results in an "Add Customer" event being sent to the OAM Server. The Event Handler then waits for the OAM Server to respond to this event with either an ID number for the new customer or and error message. The Event Handler may also receive unsolicited events from the OAM Server such as error messages. Upon receipt of such messages, the Event Handler should display the event on the Operator Terminal by using User Interface Forms.

3.2.5 Billing Daemon

Billing Daemon	
Collaborator	**Responsibility**
• Dispatcher • Connection Worker	The Billing Daemon will contact that dispatcher and send a print all bills message, specifying the period as the last day it was run to yesterday's date (inclusive). It is set up so that it can run as a scheduled automated job. (in UNIX, it could be a cron job)

The Billing Daemon is designed primarily to be an input source of regularly scheduled events. It is envisioned that of course all the bills will not get printed on a given day, but the Billing Daemon could start up on given days and tell a Connection Worker to print all bills that have a billing date of today, or to print all bills in Exchange number 1. It is expected that this functionality will be implemented in a further release of the product and not the initial release.

3.2.6 CU Handler

CU Handler	
Collaborator	**Responsibility**
• Dispatcher • CU Simple Operations • Decision Maker	The CU Handler is the central point for communication to the CU. The CU Handler is to implement Message queues to talk directly to the CU. There is a one to one relationship between each CU and a CU Handler. The Simple CU Operations will communicate to the CU Handler, and the CU Handler will notify the Dispatcher of any errors that may occur.

This module is both a Server and a Client. The Design decision to make this a component stems from the ability for the system to be distributed across many machines. Since there is only one CU, there must be a 1-1 relationship with a piece of software to receive and write messages on the event queue with the CU. It is envisioned that this component could be running on the machine that the CU is connected to and many Dispatchers and Connection Workers could access it from various points in the network. This central point to talk to the CU allows simulation to occur easily. [There's more than one CU but the point is valid]

4.0 Class Specifications

4.1 Basic System Objects

Basic system objects have been presented here that are used throughout the system and are referenced in the methods for each individual component. There is a checksum on the objects that can be modified. The checksum is used much like a cookie, to ensure that the Operator updating the data has the most current version and that no one between the time they fetched the data and when the time they update it no one has modified it.

4.1.1 Customer

```
struct Customer
{
    int    m_id;
    int    m_checkSum;
    int    m_status;
    string m_firstName;
    string m_lastName;
    string m_email;
    string m_address;
    string m_city;
    int    m_province
    string m_postalCode;
};
```

4.1.2 Subscription

```
struct Subscription
{
    int    m_id;
    int    m_customerId;
    int    m_checkSum;
    int    m_status;
    int    m_planID;
    bool   m_canReceiveCalls;
    bool   m_canCallLocal;
    bool   m_canCallLongDistance;
    string m_address;
    string m_city;
    string m_province;
    string m_postalCode;
    int    m_dialedNumber
    int    m_exchange;
    int    m_shelf;
    int    m_slot;
};
```

4.1.3 Exchange

```
struct Exchange
{
    int    m_exNum;
    string m_location;
    int    m_checkSum;
};
```

4.1.4 Equipment and Cards

```
struct Equipment
{
    byte m_exNum;
    byte m_shelf;
    byte m_slot;
    byte m_dn;
    byte m_cos;
    int  m_stat;
    int  m_checkSum;
};
struct LineCard : public Equipment {};
struct TrunkCard : public Equipment {};
```

4.1.5 Rate

```
struct Rate
{
    byte   m_srcExNum;
    byte   m_dstExnum;
    double m_rate;
    int    m_checksum;
};
```

4.1.6 Charges and Calls

```
struct Charge
{
    int    m_customerId;
    int    m_subscriptionId;
    int    m_id;
    string m_start;
    string m_finish;
    double m_amount;
    int    m_type;
};

struct Call : public Charge
{
    int    m_startTime;
    int    m_duration;
    byte   m_numberCalled;
    int    m_plan;
    double m_
};
```

4.1.7 Bill

```
struct Bill
{
    int     m_customerId;
    int     m_subscriptionId;
    int     m_exchange;
    string  m_start;
    string  m_finish;
    Charge *m_serviceCharges;
    Call   *m_calls;
    int     m_numServiceCharges;
    int     m_numCalls;
    double  m_totalLocalMin;
    double  m_totalLDMin;
    double  m_totalLDCharges;
    double  m_totalServiceCharges;
    double  m_total;
    double  m_totalSavings;
};
```

4.1.8 Error Codes

These error codes are used for errors exported in the external interfaces. For example, the XML messaging specifies an error can be present in requests. The error is an integer for internationalization purposes. It allows us to use the same backend server, but different front ends, so the backend is not locale specific.

```
enum ErrorCode
{
   MALFORMED_REQUEST=1,  CU_CONNECT_ERROR,     DATE_BEFORE_TODAY,
   DB_CONNECT_ERROR,     DN_IN_USE,            EMPTY_VARIABLE,
   INVALID_PARAMS,       INVALID_PASSWORD,     INVALID_CHECKSUM,
   INVALID_DATE,         INVALID_COS,          INVALID_STAT,
   INVALID_DN,           INVALID_RATE,         INVALID_TIME,
   INVALID_PHONE_NUMBER, INVALID_POSTAL_CODE,  INVALID_EMAIL,
   INVALID_PROVINCE,     NO_AUTHORIZATION,     NO_SUCH_CUST,
   NO_SUCH_SUB,          NO_SUCH_CALL_PLAN,    NO_SUCH_SHELF,
   NO_SUCH_SLOT,         NO_SUCH_EXCHANGE,     SYSTEM_ERROR,
   USER_NOT_FOUND
};
```

4.1.9 Status Enumeration

The XML DTD also specifies that Customers will have statuses for the Subscription data blocks. The following enumeration is applied to those fields.

```
Enum StatusCodes
{
   CANCELLED = 1, ACTIVE, SUSPENDD
};
```

4.2 The Dispatcher Process

Usage

```
dispatcher [-p port]
```

Purpose

Initialize communication then start receiving and handling messages.

Arguments

port – Port on which the dispatcher will bind and receive TCP/IP messages. (default 8723)

Implementation Notes/Pseudocode

```
main(int argc, char *argv[]) {
    Verify argument format and number of arguments
    Set port number, if one was given otherwise use the default
    Call InitCommunications with the port number.
    If initialization was successful then
```

```
        Call AcceptCommunications to begin receiving and handling messages
    }
```

4.2.1 Dispatcher Methods

4.2.1.1 AcceptCommunications()

Declaration

```
void AcceptCommunications();
```

Purpose

Continuously receives and handles messages on the previously bound socket.

Implementation Notes/Pseudocode

```
for (;;) {
    accept a message on the previously bound socket
    If the message received is an error then
        Send the message to all active Connection Workers using their pipe.
    Else If it is a client request then
        Spawn a new Connection Worker, cw
        Handoff all further communication with the client to cw
        Pass cw the client’s host name (determined from socket connection)
        Pass cw a handle to a new pipe, p
        Add cw,p to the list of active Connection Workers and their pipe
}
```

Notes:

1) Depending on how ‘Spawn’ is implemented the socket connection may have to be closed after spawning the connection worker (e.g. this will be the case if a Unix fork is used).

2) This method will only return if a signal interrupt (Ctrl-C) is detected from the input console or if a fatal error occurs.

4.2.1.2 InitCommunications()

Declaration

```
bool InitCommunications(short port);
```

Purpose

Initialize communication for the dispatcher.

Arguments

port – the port number to bind on.

Return Value

True if successful, false otherwise.

Implementation Notes/Pseudocode

Bind to a TCP/IP socket on the specified port (on all network adapters).

4.3 The Connection Worker Process

Usage

```
conworker <socketHandle> <clientHostname> <pipeHandle>
conworker –F
```

Purpose

Initialize communication then start receiving and handling messages from a client or from the Dispatcher

Arguments

socketHandle – socket to use for all client communication
clientHostname – host name of client machine (used for logging)
pipeHandle – alternate communication channel for messages from the Dispatcher
–F – use standard i/o for communication (used for interactive debugging)

Implementation Notes/Pseudocode

```
main(int argc, char *argv[]) {
    Verify argument format and number of arguments
    If –F flag specified then
```

```
        p = some default value
        chost = some default value
        in = standard input
       out = standard output
    else
        p = Get pipe handle from command-line
        chost = Get client host from command-line
        Get socket handle from command-line
        in = socket handle’s receive channel
        out = socket handle’s send channel
    Call InitCommunications(p)
    If initialization was successful then
       Call DecisionMaker::AcceptCommunications (in,out)
}
```

4.3.1 Connection Worker Methods

4.3.1.1 InitCommunications()

Declaration

```
bool InitCommunications(int pipeHandle);
```

Purpose

Initialize communication for the Connection Worker.

Arguments

pipeHandle – messages from the Dispatcher will be received on this pipe.

Return Value

True if successful, false otherwise.

Implementation Notes/Pseudocode

Load the TCP/IP libraries if required.

4.4 The Decision Maker Class

4.4.1 Decision Maker Methods

4.4.1.1 AcceptCommunications()

Declaration

```
void AcceptCommunications(Stream* in, Stream* out);
```

Purpose

Continuously processes requests from a client or from the Dispatcher.

Arguments

in – an open stream from which message from the client will be read.
out – an open stream to which messages to the client will be written.
Note: See the External Interfaces section for details on the format of a message.

Implementation Notes/Pseudocode

```
For(;;) {
   Read a request
   Parse the request to validate its structure
   d = Create a new XML document
   Determine the Complex Operation that corresponds to this request
   Determine what data this Complex Operation needs from the request, c
   Call the Complex Operation passing c and d.
   Write d to the output stream
}
```

Notes:

1) The request is validated using the DTD specified in the External Interfaces section.
2) This method will only exit if a fatal error occurs or when the communications channel to the

client is dropped.

4.5 The O&A Operations Class

All methods in the following subsections have a similar signature format, so we will define all commonalities here. The only differences are the declaration, purpose and implementation notes of the method as well as the names of the XML elements received (the ‘head’ variable) and returned; hence, only these details will be specified in the following subsections. For more details on the XML elements, see the External Interfaces section.

Declaration `DOM_Node methodName(DOM_Document& theDoc, DOM_Node head);`
Purpose Specified in subsections
Arguments theDoc – the document that will be returned to the user (we need this to create DOM_Nodes, i.e. to construct the return value) head – represents a correctly formatted _______ element.
Return Value A correctly formatted _______ element.
Implementation Notes/Pseudocode All methods in the Complex Operations layer use methods from the Simple Operations layer. In the sections below, a list of the Simple Operations necessary to complete a given Complex Operation are listed using the following syntax: Uses simpleOp1(), simpleOp2(), … Notice that arguments are not listed for the Simple Operations.

4.5.1 General Methods

4.5.1.1 Login()

Declaration `DOM_Node Login(DOM_Document& theDoc, DOM_Node head);`
Purpose Log the user into the OAM system.
Arguments head – <LoginRequest>
Return Value <LoginResponse>
Implementation Notes/Pseudocode Uses DBSimpleOps.Connect() DBSimpleOps.Disconnect(). The username and password received from the <LoginRequest> must be stored so that the other methods in the O&A Operations class can establish a database connection. An operator’s userid and password are the same as the userid and password used to connect to the database.

4.5.2 Customer Methods

4.5.2.1 AddCustomer()

Declaration `DOM_Node AddCustomer(DOM_Document& theDoc, DOM_Node head);`
Purpose Add a customer to the OAM system.
Arguments head – <AddCustomerRequest>
Return Value <AddCustomerResponse>

Implementation Notes/Pseudocode

Uses CreateCustomer()

4.5.2.2 EditCustomer()

Declaration

```
DOM_Node EditCustomer(DOM_Document& theDoc, DOM_Node head);
```

Purpose

Edit customer information already stored in the OAM system.

Arguments

head – <EditCustomerRequest>

Return Value

<EditCustomerResponse>

Implementation Notes/Pseudocode

Uses UpdateCustomer()

4.5.2.3 FindCustomer()

Declaration

```
DOM_Node FindCustomer(DOM_Document& theDoc, DOM_Node head);
```

Purpose

Find a customer or customers that match the provide information.

Arguments

head – <FindCustomerRequest>

Return Value

<FindCustomerResponse>

Implementation Notes/Pseudocode

Uses RetrieveCustomer() and/or RetrieveSubscription()

4.5.3 Subscription Methods

4.5.3.1 AddSubscription()

Declaration

```
DOM_Node AddSubscription(DOM_Document& theDoc, DOM_Node head);
```

Purpose

Add a subscription to an existing customer.

Arguments

head – <AddSubscriptionRequest>

Return Value

<AddSubscriptionResponse>

Implementation Notes/Pseudocode

```
CreateSubscription()  // Create a subscription (allocate line card and phone number)
UpdateDB()  // Update the dialed number on the line card that was allocated
```

4.5.3.2 CancelSubscription()

Declaration

```
DOM_Node CancelSubscription(DOM_Document& theDoc, DOM_Node head);
```

Purpose

Cancel a subscription; this will release the line card and phone number associated with the subscription. Canceling a subscription also results in a bill being sent.

Arguments

head – <CancelSubscriptionRequest>

Return Value

<CancelSubscriptionResponse>

Implementation Notes/Pseudocode

```
UpdateSubscription()  // Free this subscription’s line card and phone number
UpdateDB()  //Tell the CU that the line card is free (set the card’s DN to 99)
GeneratePDF(), PrintPDFl(), EmailPDFl()  // Email and Print the bill
```

4.5.3.3 EditAllSubscriptions()

Declaration

```
DOM_Node EditAllSubscriptions(DOM_Document& theDoc, DOM_Node head);
```

Purpose

Update all subscriptions that satisfy the given criteria.

Arguments

head – <EditAllSubscriptionsRequest>

Return Value

<EditAllSubscriptionsResponse>

Implementation Notes/Pseudocode

```
UpdateAllSubscriptions()  // Update multiple subscriptions
UpdateDB()          // Update the CU if necessary (if the phone number or the class
                    // of service has changed)
```

4.5.3.4 EditSubscription()

Declaration

```
DOM_Node EditSubscription(DOM_Document& theDoc, DOM_Node head);
```

Purpose

Update an existing subscription.

Arguments

head – <EditSubscriptionRequest>

Return Value

<EditSubscriptionResponse>

Implementation Notes/Pseudocode

```
UpdateSubscription()  // Update the given Subscription
UpdateDB()          // Update the CU if necessary (if the phone number or the class
                    // of service has changed)
```

4.5.3.5 FindSubscription()

Declaration

```
DOM_Node FindSubscription(DOM_Document& theDoc, DOM_Node head);
```

Purpose

Find one or more subscriptions given partial information.

Arguments

head – <FindSubscriptionRequest>

Return Value

<FindSubscriptionResponse>

Implementation Notes/Pseudocode

Uses RetrieveSubscription()

4.5.3.6 ResumeSubscription()

Declaration

```
DOM_Node ResumeSubscription(DOM_Document& theDoc, DOM_Node head);
```

Purpose

Resume a subscription; associates a linecard with the subscription.

Arguments

head – <ResumeSubscriptionRequest>

Return Value

<ResumeSubscriptionResponse>

Implementation Notes/Pseudocode

```
UpdateSubscription()  // Find and allocate a line card for this subscription
UpdateDB()  // Update the dialed number on the line card that was allocated
```

4.5.3.7 SuspendSubscription()

Declaration

```
DOM_Node SuspendSubscription(DOM_Document& theDoc, DOM_Node head);
```

Purpose

Suspend a subscription; this will release the line card associated with a subscription (but not the phone number)

Arguments

head – <SuspendSubscriptionRequest>

Return Value

<SuspendSubscriptionResponse>

Implementation Notes/Pseudocode

```
UpdateSubscription() // Free this subscription’s line card but not its phone number
UpdateDB() // Tell the CU that the line card is free (set the card’s DN to 99)
```

4.6 The Billing Operations Class

4.6.1 Plan Methods

4.6.1.1 AddPlan()

Declaration

```
DOM_Node AddPlan(DOM_Document& theDoc, DOM_Node head);
```

Purpose

Add a new long distance plan to the OAM System.

Arguments

head – <AddPlanRequest>

Return Value

<AddPlanResponse>

Implementation Notes/Pseudocode

Uses CreatePlan()

4.6.1.2 ActivatePlan()

Declaration

```
DOM_Node ActivatePlan(DOM_Document& theDoc, DOM_Node head);
```

Purpose

Sets the given plans status to active.

Arguments

head – <ActivatePlanRequest>

Return Value

<ActivatePlanResponse>

Implementation Notes/Pseudocode

Uses UpdatePlan()

4.6.1.3 DeactivatePlan()

Declaration

```
DOM_Node DeactivatePlan(DOM_Document& theDoc, DOM_Node head);
```

Purpose

Receives an old plan and a replacement plan. Sets the old plan’s status to inactive. All subscriptions that reference the old plan are switched to the replacement plan.

Arguments

head – <ActivatePlanRequest>

Return Value

<ActivatePlanResponse>

Implementation Notes/Pseudocode

Uses UpdatePlan(), UpdateAllSubscriptions()

4.6.1.4 DescribePlan()

Declaration

```
DOM_Node DescribePlan(DOM_Document& theDoc, DOM_Node head);
```

Purpose

Retrieve the information pertaining for a given plan.

Arguments

head – <DescribePlanRequest>

Return Value

<DescribePlanResponse>

Implementation Notes/Pseudocode

Uses RetrievePlan()

4.6.2 Bill Methods

4.6.2.1 DescribeBill()

Declaration

```
DOM_Node DescribeBill(DOM_Document& theDoc, DOM_Node head);
```

Purpose

Retrieves information pertaining the given subscription and period (i.e. a list of changes).

Arguments

head – <DescribeBillRequest>

Return Value

<DescribeBillResponse>

Implementation Notes/Pseudocode

Uses RetrieveBill()

4.6.2.2 OutputAllBills()

Declaration

```
DOM_Node OutputAllBills(DOM_Document& theDoc, DOM_Node head);
```

Purpose

Outputs a bill for every subscription with an outstanding balance.

Arguments

head – <OutputAllBillsRequest>

Return Value

<OutputAllBillsResponse>

Implementation Notes/Pseudocode

Uses RetrieveBill(), GeneratePDF(), EmailPDF(), PrintPDF()

4.6.2.3 OutputBill()

Declaration

```
DOM_Node OutputBill(DOM_Document& theDoc, DOM_Node head);
```

Purpose

Outputs the bill for a given subscription and period.

Arguments

head – <OutputBillRequest>

Return Value

<OutputBillResponse>

Implementation Notes/Pseudocode

Uses RetrieveBill(), GeneratePDF(), EmailPDF(), PrintPDF()

4.6.3 Charge and Call Methods

4.6.3.1 AddCharge()

Declaration

```
DOM_Node AddCharge(DOM_Document& theDoc, DOM_Node head);
```

Purpose

Adds a charge to the given subscription.

Arguments

head – <AddChargeRequest> or <BillCallRequest>

Return Value

<AddChargeResponse> or <BillCallResponse>

Implementation Notes/Pseudocode

```
If <BillCallRequest> is received then
    c =  a new Call object based on the information provided by the element
Else if <AddChargeRequest> is received then
```

```
    c = a new Charge object based on the information provided by the element
Call CreateCharge(c)
```

4.6.3.2 AddChargeAll()

Declaration

```
DOM_Node AddChargeAll(DOM_Document& theDoc, DOM_Node head);
```

Purpose

Adds a charge to all subscriptions that satisfy given criteria.

Arguments

head – <AddChargeAllRequest>

Return Value

<AddChargeAllResponse>

Implementation Notes/Pseudocode

Uses RetrieveSubscription(), RetrieveBill() and CreateCharge()

4.7 The Maintenance Operations Class

4.7.1 Exchange Methods

4.7.1.1 AssociateExchange() and DisassociateExchange()

Declaration

```
DOM_Node AssociateExchange(DOM_Document& theDoc, DOM_Node head);
```

and

```
DOM_Node DisassociateExchange(DOM_Document& theDoc, DOM_Node head);
```

Purpose

Allows one exchange to call another. This means allocating a trunk card within the source exchange that is associated with a remote or destination exchange. Associating to a remote exchange will allow calls to be routed to that exchange. Disassociating from a remote exchange will disassociate the trunk card and the remote exchange.

Arguments

head – <AssociateExchangeRequest> and <DisassociateExchangeRequest>

Return Value

<AssociateExchangeResponse> and <DisassociateExchangeResponse>

Implementation Notes/Pseudocode

```
UpdateEquipment() // Allocate a trunkcard and set its dialed number
                  // to the number of the remote exchange
UpdateDB() // Inform the CU of the change to the trunk card
```

4.7.1.2 DescribeExchange()

Declaration

```
DOM_Node DescribeExchange(DOM_Document& theDoc, DOM_Node head);
```

Purpose

Retrieves information about one or all exchanges.

Arguments

head – <DescribeExchangeRequest>

Return Value

<DescribeExchangeResponse>

Implementation Notes/Pseudocode

Uses RetrieveExchange()

4.7.1.3 EditExchange()

Declaration

```
DOM_Node EditExchange(DOM_Document& theDoc, DOM_Node head);
```

Purpose

Update the information stored for a given exchange.

Arguments

head – <EditExchangeRequest>

Return Value <EditExchangeResponse>
Implementation Notes/Pseudocode Uses UpdateExchange()

4.7.2 Equipment Methods

4.7.2.1 DescribeEquipment()

Declaration `DOM_Node DescribeEquipment(DOM_Document& theDoc, DOM_Node head);`
Purpose Retrieve the information pertaining to one or more cards. This information is retrieve from the OAM's version of the exchange database.
Arguments Head – <DescribeCardRequest>
Return Value <DescribeCardResponse>
Implementation Notes/Pseudocode Uses RetrieveCard()

4.7.2.2 EnableEquipment() and DisableEquipment()

Declaration
`DOM_Node EnableEquipment(DOM_Document& theDoc, DOM_Node head);`
and
`DOM_Node DisableEquipment(DOM_Document& theDoc, DOM_Node head);`

Purpose Requests that the CU change the status of a given piece of equipment to Enabled or Disabled.
Arguments head – <EnableEquipmentRequest> and <DisableEquipmentRequest>
Return Value <EnableEquipmentResponse> and <DisableEquipmentResponse>
Implementation Notes/Pseudocode Uses SetStatus(). Does not call UpdateEquipment() because the CU will send an update request to acknowledge that it has handled our SetStatus() request.

4.7.2.3 QueryCU()

Declaration `DOM_Node QueryCU(DOM_Document& theDoc, DOM_Node head);`
Purpose Retrieve the information pertaining to a card. This information is retrieve from the CU's version of the exchange database.
Arguments Head – <QueryCURequest>
Return Value <QueryCUResponse>
Implementation Notes/Pseudocode Uses QueryCU()
Implementation Notes/Pseudocode Uses TestEquipment()

4.7.2.4 ResetEquipment()

Declaration `DOM_Node ResetEquipment(DOM_Document& theDoc, DOM_Node head);`
Purpose Requests that the CU reset a given card.
Arguments head – <ResetEquipmentRequest>
Return Value

<ResetEquipmentResponse>
Implementation Notes/Pseudocode Uses ResetEquipment()

4.7.2.5 TestEquipment()

Declaration `DOM_Node TestEquipment(DOM_Document& theDoc, DOM_Node head);`
Purpose Requests that the CU test a given piece of equipment.
Arguments Head – <TestEquipmentRequest>
Return Value <TestEquipmentResponse>

4.8 The Database Simple Operations Class

4.8.1 Commonalities in Methods

All methods that receive a filter must be implemented using dynamic SQL statements. Here is a brief example of what is meant by ‘dynamic SQL statements’

```
s = “SELECT * FROM Customer WHERE ”
If (filter & CUSTOMER_FIRSTNAME) then
        s = s + “firstName=” + c.firstName
if …
```

Many of the methods define in the flowing subsections are very similar. These similarities will be define once in the following tables

4.8.1.1 Create*Object*()

Declaration `void CreateObject(Object* newObject);`
Purpose Create a new record in the _____ table of the database using the given Object. Requires a connection to the database. An exception will be thrown if an error occurs. See the actually method specifications for the name of the table into which the new record is inserted.
Arguments new*Object* – the *Object* whose information will be used to create the new record in the database.

4.8.1.2 Retrieve*Object*()

Declaration `void RetrieveObject(Object* criteria, unsigned int filter, Object**resultArray, int* resultSize);`
Purpose Retrieves 0 or more *Objects* that match the given a search criteria. The calling method is a responsible for deallocating the resultArray. Requires a connection to the database. An exception will be thrown if an error occurs.
Arguments criteria – An *Object* whose values will be used as search criteria when querying the database . filter – Determines which fields in the given *Object* should be used as search criteria. resultArray – Used to return an array of *Objects* that match the given criteria. resultSize – Used to return the number of records that matched the given criteria.

4.8.1.3 Update*Object*()

Declaration `void UpdateObject(Object* alteredObject, unsigned int filter);`

Purpose

Updates the record whose unique identifiers match those of the given *Object* (i.e. this method is allow to effect at most one record). If the given *Object* has a checkSum then before the update occurs this checkSum is checked to make sure it represents the most recent version of the record corresponding to this object. Requires a connection to the database. An exception will be thrown if an error occurs.

Arguments

altered*Object* – The *Object* containing the updated information
filter – Used to specify which fields in the *Object* contain updated information

4.8.2 Connection Methods

4.8.2.1 Connect()

Declaration

```
void Connect(char* dbName, char* host, char* userId, char*
userPassword, int port);
```

Purpose

Connect to a given database on a particular host and port. The user id and password will be that of the Operator, so they will be granted specific access into the database. An exception will be thrown if an error occurs.

Arguments

dbName – the name of the database to connect to
host – the host name of the machine that the data server runs on
userID, userPassword – the user id and password of to use to connect to the database
port – the port that the data server is bound to

Implementation Notes/Pseudocode

This method must be repeatable so that a DBSimpleOps object can be reused.

4.8.2.2 Disconnect()

Declaration

```
void Disconnect();
```

Purpose

Disconnect from the current database. An exception will be thrown if an error occurs.

Implementation Notes/Pseudocode

Calling this method without a current connect will not cause an error.

4.8.2.3 IsConnected()

Declaration

```
Bool IsConnected();
```

Purpose

Determine if there is a current database connection.

Returns

True if there is a connection, false otherwise.

4.8.3 Customer Methods

4.8.3.1 CreateCustomer()

Declaration

```
void CreateCustomer(Customer* newCustomer);
```

Purpose

Create a new record in the Customer table, using newCustomer.

Arguments

newCustomer – if the record is successfully created then customer id of this object will be set. Also see “Commonalities in Methods” section.

4.8.3.2 RetrieveCustomer()

Declaration

```
void RetrieveCustomer(Customer* criteria, unsigned int filter,
Customer** resultArray, int* resultSize);
```

Purpose

Retrieves 0 or more Customers that match the given a search criteria.

Arguments

See “Commonalities in Methods” section.

4.8.3.3 UpdateCustomer()

Declaration

```
void UpdateCustomer(Customer* alteredCustomer, unsigned int
filter);
```

Purpose

Updates the customer record whose customer id matches the id of the given Customer.

Arguments

See “Commonalities in Methods” section.

4.8.4 Subscription Methods

4.8.4.1 CreateSubscription()

Declaration

```
void CreateSubscription(Subscription* newSubscription);
```

Purpose

Create a new record in the Subscriptions table, using newSubscription.

Arguments

newSubscription – The customer id of this object must contain a valid customer id. If a subscription is successfully created then this object’s subscription id will be set.

Also see “Commonalities in Methods” section.

Implementation Notes/Pseudocode

```
Insert the given info (excluding the phone number) into the Subscription Table
If a phone number has been specified
  If the phone number is already in use within the new subscription’s exchange
      Throw an exception
Else
  Generate a new phone number that is unique within this subscription’s exchange
     i.e. choose one of the values returned by
           SELECT dialedNumber
           FROM PossibleDN
           WHERE dn NOT IN (SELECT dialedNumber
                            FROM Subscription as S
                            WHERE S.exchange = newSubscription.m_exchange)
Update the dialed number of the new subscription’s record
Atomically Allocate a functional (i.e. status = 000*), free line card (i.e. DN=99)
        i.e. let COS = (newSubscription.m_canCallLongDistance & 0x01) << 3
                     | (newSubscription.m_canCallLocal & 0x01) << 2
                     | (newSubscription.canReceiveCalls & 0x01)
        UPDATE Card
        SET(dialedNumber, classOfService)
        VALUES (newSubscription.m_dn, COS)
        WHERE dialedNumber=99 AND status IS LIKE ‘000%’
If no such line cards exist (rows effected by UPDATE is 0) then throw an exception
Call UpdateDB() to inform the CU of the changes to the allocated line card
```

4.8.4.2 RetrieveSubscription()

Declaration

```
void RetrieveSubscription(Subscription* criteria, unsigned int
filter, Subscription** resultArray, int* resultSize);
```

Purpose

Retrieves 0 or more Subscriptions that match the given a search criteria.

Arguments

See “Commonalities in Methods” section.

4.8.4.3 UpdateAllSubscriptions()

Declaration

```
void UpdateAllSubscriptions(Subscription* criteria, unsigned int
searchFilter, Subscription* alteredSubscription, unsigned int
setFilter, int* recordsAltered);
```

Purpose

Update the subscription records that meet the given criteria.

Arguments

criteria – A Subscription object whose values will be used as search criteria when selecting records to be updated.

searchFilter – Determines which fields in the given criteria should be used as search criteria.

alteredSubscription – The Subscription object containing the updated information

filter – Used to specify which fields of alteredSubscription will be used to update the selected rows.

recordsAltered – Used to return the number of subscriptions that were updated as a result of this method.

4.8.4.4 UpdateSubscription()

Declaration

```
void UpdateSubscription(Subscription* alteredSubscription, unsigned
int filter);
```

Purpose

Updates the subscription record whose customer id and subscription id match those of the given Subscription.

Arguments

See “Commonalities in Methods” section.

4.8.5 Exchange Methods

4.8.5.1 CreateExchange()

Declaration

```
void CreateExchange(Exchange* newExchange);
```

Purpose

Creates a new exchange record in the Exchange table based on newExchange. Also initialize the Equipment table with entries for each slot in the exchange and inserts record into the Rate table for calls between the given exchange and all existing exchanges (plus one more record for the local rate).

Arguments

newExchange – the exchange number of this object must be set to a valid exchange number that does not already exist in the database.

Also see “Commonalities in Methods” section.

Implementation Notes/Pseudocode

Refer to the Software Interface Description for details about how many shelves and slots an exchange has, as well as the default values for each equipment type. All rates are stored in the CallingRates table, a local rate is stored as an entry with the same value for source and destination exchange. Use CreateRate() and CreateEquipment() to implement this method.

4.8.5.2 RetrieveExchange()

Declaration

```
void RetrieveExchange(Exchange* criteria, unsigned int filter,
Exchange** resultArray, int* resultSize);
```

Purpose

Retrieves 0 or more Exchanges that match the given a search criteria.

Arguments

See “Commonalities in Methods” section.

4.8.5.3 UpdateExchange()

Declaration

```
void UpdateExchange(Exchange* alteredExchange, unsigned int
filter);
```

Purpose

Updates the exchange record whose exchange number matches that of the given Exchange.

Arguments

See “Commonalities in Methods” section.

4.8.6 Rate Methods

4.8.6.1 CreateRate()

Declaration

```
void CreateRate(Rate* newRate);
```

Purpose

Creates a new rate record in the Rate table based on newRate.

Arguments

See “Commonalities in Methods” section.

4.8.6.2 RetrieveRate()

Declaration

```
void RetrieveRate(Rate* criteria, unsigned int filter, Rate**
resultArray, int* resultSize);
```

Purpose

Retrieves 0 or more Rates that match the given a search criteria.

Arguments

See “Commonalities in Methods” section.

4.8.6.3 UpdateRate()

Declaration

```
void UpdateRate(Rate* alteredRate, unsigned int filter);
```

Purpose

Updates the rate record whose source and destination exchange match those of the given Rate.

Arguments

See “Commonalities in Methods” section.

Note that the filter argument is not necessary here however it is included for consistency with other methods, and for any future expansions to the Rate object

4.8.7 Equipment Methods

4.8.7.1 CreateEquipment()

Declaration

```
void CreateEquipment(Equipment* newEquipment);
```

Purpose

Creates a new equipment record in the Equipment table based on newEquipment.

Arguments

newEquipment – the exchange, shelf and slot of this object must be set.

Also see “Commonalities in Methods” section.

Implementation Notes/Pseudocode

Refer to the Software Interface Description for details about the default values for each equipment type.

4.8.7.2 RetrieveEquipment()

Declaration

```
void RetrieveEquipment(Equipment* criteria, unsigned int filter,
Equipment** resultArray, int* resultSize);
```

Purpose

Retrieves 0 or more pieces of Equipment that match the given a search criteria.

Arguments

See “Commonalities in Methods” section.

4.8.7.3 UpdateEquipment()

Declaration

```
void UpdateEquipment(Equipment* alteredEquipment, unsigned int
filter);
```

Purpose

Updates the equipment record whose exchange number, shelf and slot match those of the given Equipment object.

Arguments

See “Commonalities in Methods” section.

4.8.8 Bill Methods

4.8.8.1 RetrieveBill()

Declaration

```
void RetrieveBill(Bill* criteria, unsigned int filter, Bill**
resultArray, int* resultSize);
```

Purpose

Retrieves 0 or more Bills that match the given a search criteria.

Arguments

See “Commonalities in Methods” section.

Implementation Notes/Pseudocode

Use RetrieveCharge() and any queries necessary to calculate totals stored in the Bill object.

4.8.9 Charge Methods

4.8.9.1 CreateCharge()

Declaration

```
void CreateCharge(Charge* newCharge);
```

Purpose

Creates a new record in the Charge table based on newCharge. A record will also be inserted into the Call table if newCharge is an instance of the Call class.

Arguments

newCharge – the customer id and subscription id of this object must be set. If the new record is successfully created then the charge id of this object will be set. If this object is an instance of the Call class then this object’s plan id will also be set.

Also see “Commonalities in Methods” section.

Implementation Notes/Pseudocode

```
Insert the charge into the Charge table and retrieve the new records charge id
Set newCharge’s charge id to the retrieved value
If newCharge is an instance of the Call class
    Get the plan that corresponds to this charge’s subscription
    Set newCharge’s plan id
    Insert a new record into the Call table based on newCharge
```

4.8.9.2 RetrieveCharge()

Declaration

```
void RetrieveCharge(Charge* criteria, unsigned int filter, Charge**
resultArray, int* resultSize);
```

Purpose

Retrieves 0 or more charges that match the given a search criteria.

Arguments

resultArray – may contain instances of both the Charge and Call classes.
Also See “Commonalities in Methods” section.

4.8.10 Plan Methods

4.8.10.1 CreatePlan()

Declaration

```
void CreatePlan(Plan* newPlan);
```

Purpose

Creates a new record in the Plan table based on newPlan.

Arguments

newPlan – If the new record is successfully created then the plan id of this object will be set.
Also see “Commonalities in Methods” section.

4.8.10.2 RetrievePlan()

Declaration

```
void RetrievePlan(Plan* criteria, unsigned int filter, Plan**
resultArray, int* resultSize);
```

Purpose

Retrieves 0 or more plans that match the given a search criteria.

Arguments

See “Commonalities in Methods” section.

4.8.10.3 UpdatePlan()

Declaration

```
void UpdatePlan(Plan* alteredPlan, unsigned int filter);
```

Purpose

Updates the plan record whose plan id matches that of the given Plan.

Arguments

See “Commonalities in Methods” section.

4.9 The CU Simple Operations Class

4.9.1 Connection Methods

4.9.1.1 Connect()

Declaration

```
void Connect(char* host, short port);
```

Purpose

Connect to a CU Handler at the specified host and port. An exception will be thrown if an error occurs.

Implementation Notes/Pseudocode

This method must be repeatable so that a CUSimpleOps object can be reused.

4.9.1.2 Disconnect()

Declaration

```
void Disconnect();
```

Purpose

Disconnect from a previously connect CU Handler. An exception will be thrown if an error occurs.

Implementation Notes/Pseudocode

If there is no current connection then this method should have no effect.

4.9.2 Maintenance and Debugging Methods

4.9.2.1 QueryDB()

Declaration

```
void QueryDB(Equipment &equipment, Equipment* result);
```

Purpose

Asks the CU (via the CU Handler) to return its copy of the information stored from a given piece of equipment. Requires a connection to a CUHandler. An exception will be thrown if an error occurs.

Arguments

equipment – The piece of equipment to be reset.
result – Object that stores the information of the equipment specified

Implementation Notes/Pseudocode

Format and send a QueryDB message to the CU Handler. See the Software Interface Description Document for more details

4.9.2.2 ResetEquipment()

Declaration

```
void ResetEquipment(Equipment &equipment);
```

Purpose

Asks the CU (via the CU Handler) to reset a given piece of equipment. Requires a connection to a CUHandler. An exception will be thrown if an error occurs.

Arguments

equipment – The piece of equipment to be reset.

Implementation Notes/Pseudocode

Format and send a ResetDevice message to the CU Handler. See the Software Interface Description Document for more details

4.9.2.3 SetStatus()

Declaration

```
void SetStatus(Equipment &equipment);
```

Purpose

Asks the CU (via the CU Handler) to change the status of the given piece of equipment. Requires a connection to a CUHandler. An exception will be thrown if an error occurs.

Arguments

equipment – The piece of equipment whose status is to be updated.

Implementation Notes/Pseudocode

Format and send a SetStatus message to the CU Handler. See the Software Interface Description Document for more details

4.9.2.4 TestEquipment()

Declaration

```
void TestEquipment(Equipment &equipment);
```

Purpose

Asks the CU (via the CU Handler) to test the given piece of equipment. Requires a connection to a CUHandler. An exception will be thrown if an error occurs.

Arguments

equipment – The piece of equipment to be tested.

Implementation Notes/Pseudocode

Format and send a TestEN message to the CU Handler. See the Software Interface Description Document for more details

4.9.2.5 UpdateDB()

Declaration

```
void UpdateDB(Equipment &alteredEquipment);
```

Purpose

Tells the CU (via the CU Handler) to update its copy of the Equipment table. Requires a connection to a CUHandler. An exception will be thrown if an error occurs.

Arguments

alteredEquipment – The updated data to be included in the message to the CU

Implementation Notes/Pseudocode

Format and send an UpdateDB to the CU Handler. See the Software Interface Description Document for more details

4.10 The Output Simple Operations Class

4.10.1 Output Methods

4.10.1.1 EmailPDF()

Declaration

```
void EmailPDF(Customer* cust, Subscription* sub, char
pdfFile[100]);
```

Purpose

Email the given PDF file to the given customer. An exception will be thrown if an error occurs.

Arguments

cust – The customer information that corresponds to the bill represented by the given PDF file
sub – The subscription information that corresponds to the bill represented by the given PDF file
pdfFile – the path to a PDF file representing a bill

4.10.1.2 GeneratePDF()

Declaration

```
void GeneratePDF(Customer* cust, Subscription* sub, Bill* bill,
char pdfFile[100]);
```

Purpose

Generate a PDF file representing the bill for the given customer and subscription. The method will create a file in the temporary disk space on the machine. The calling method is responsible for deleting the file by calling ReleasePDF. An exception will be thrown if an error occurs.

Arguments

cust – The customer information that corresponds to the given bill
sub – The subscription information that corresponds to the given bill
bill – The billing information to include in the generated file
buffer – Used to return the path to a PDF file

Arguments

cust – The customer information that corresponds to the given bill
sub – The subscription information that corresponds to the given bill
bill – The billing information to include in the generated file
buffer – Used to return the path to a PDF file

4.10.1.3 PrintPDF()

Declaration

```
void PrintPDF(char* outputDevice, Customer* cust, Subscription*
sub, char pdfFile[100]);
```

Purpose

Print the given PDF file using the given output device. An exception will be thrown if an error occurs.

Note: The printing may not occur immediately, the PrintPDF method may pass the bill off to a Print Server or Print Queue to be printed

Arguments

outputDevice – An identifier of a device that can be used to print the given PDF file
cust – The customer information that corresponds to the bill represented by the given PDF file
sub – The subscription information that corresponds to the bill represented by the given PDF file
pdfFile – the path to a PDF file representing a bill

4.10.1.4 ReleasePDF()

Declaration

```
void ReleasePDF(char pdfFile[100]);
```

Purpose

Release a previously allocated PDF file. This is accomplished by deleting the file specified.

Arguments

pdfFile – the path to a PDF file representing a bill

4.11 The CU Handler Process

Usage

```
cuhandler dispatcherHost dispatcherPort eventQueueHandle
         [-p IncomingPort]
```

Purpose

Initialize communication then start receiving and handling messages from a client or from the Dispatcher

Arguments

dispatcherHost – host name of machine running the Dispatcher process
dispatcherPort – port number of that the Dispatcher is bound to
eventQueueHandle – a handle to where the event queue should be created to talk with a CU
IncomingPort – Port on which the cuhandler will bind and receive TCP/IP messages (default 8724)

Implementation Notes/Pseudocode

```
main(int argc, char *argv[]) {
   Verify argument format and number of arguments
   Set port number, if one was given otherwise use the default
   Call InitCommunications with the dispathcer host and port, port number and event queue
   If initialization was successful then
      Call StartCUListener to begin receiving messages from the CU's
      Call AcceptCommunications to begin receiving and handling
           messages from other clients (Connection Workers)
}
```

4.11.1 CU Handler Methods

4.11.1.1 AcceptCommunications()

Declaration

```
void AcceptCommunications();
```

Purpose

Continuously receives and handles messages on the previously bound socket.

Implementation Notes/Pseudocode

```
for (;;) {
   accept a message on the previously bound socket
   Reformat the message if necessary
   Write the message to the Message Queue
}
```

Notes:

1) This method will only return if a signal interrupt (Ctrl-C) is detected from the input console or if a fatal error occurs.

4.11.1.2 InitCommunications()

Declaration

```
bool InitCommunications(char* requestedHost, short port, short
listernPort, char* eventQueueHandle);
```

Purpose

Initialize communication for the CU Handler.

Arguments

RequestedHost, port – Establish a socket connection to the OAM Server on this remote system
listenPort – Listen for messages on this port
eventQueueHandle - a handle to where the event queue should be created to talk with a CU

Return Value

True if successful, false otherwise.

Implementation Notes/Pseudocode

Bind to a TCP/IP socket on the listen port (on all network adapters). Then open a second socket (the send socket to the OAMServer) to the requestedHost and port. Finally send a login request on the send socket.

4.11.1.3 StartCUListener ()

Declaration

```
void StartCUListener(char* queueName);
```

Purpose

Create a MessageQueue and a new thread of control to receive information from the CU.

Arguments

queueName – The queue that is created will be given this name.

Implementation Notes/Pseudocode

```
Create the message queue
Start a new thread to execute the following:
for (;;) {
    wait on the queue for a message, m
    if m is and UpdateDB, BillCall message then
        Send a message on the send socket (to the Connection Worker)
    if m is an UpdateDB message and it represents a equipment failure then
        Create a new connection to the OAM Server
        Send an error message to the OAM Server (received by the dispatcher)
        Close this connection
}
```

4.12 The Billing Daemon

Usage

```
billdaemon dispatcherHost dispatcherPort
```

Purpose

Notify the OAM Server when it is time to output all bills

Arguments

dispatcherHost – host name of machine running the Dispatcher process
dispatcherPort – port number of that the Dispatcher is bound to

Implementation Notes/Pseudocode

```
main(int argc, char *argv[]) {
    Verify argument format and number of arguments
    Connect to the OAM Server
    Send a <GenerateAllBillsRequset> to the OAM Server
    If an error is returned Disconnect and exit
    Send an <OutputAllBillsRequest> to the OAM Server
    Disconnect
}
```

4.13 The OAM Client Process

4.13.1 UI Forms

Declaration

```
proc DisplayAddEditCustomerScreen { frame, isAdd }
```

Purpose

Display the screen used to add and edit a customer’s information. Display the form in the given frame. The widgets used solely for editing will be disabled if the isAdd variable is true.

Declaration

```
proc DisplayAddEditSubscriptionScreen { frame, isAdd }
```

Purpose

Display the screen used to add or edit a subscription’s information. Display the form in the given frame. The widgets used solely for editing will be disabled if the isAdd variable is true.

Declaration

```
proc DisplayCallPlanScreen { frame }
```

Purpose

Display the call plan screen in the given frame.

Declaration

```
proc DisplayCustomerScreen { frame }
```

Purpose

Display the customer screen in the given frame.

Declaration

```
proc DisplayDebugConsole { frame }
```

Purpose

Display the debug console in the given frame.

Declaration

```
proc DisplayError { anErrorMessage }
```

Purpose

Display the given error message.

Declaration

```
proc DisplayExchangeDetails { frame }
```

Purpose

Display the exchange details screen in the given frame.

Declaration

```
proc DisplayLoginScreen { frame }
```

Purpose

Display the login screen in the given frame.

Declaration

```
proc DisplayMain { frame }
```

Purpose

Display the main form in the given frame.

Declaration

```
proc DisplaySubscriptionScreen { frame }
```

Purpose

Display the subscription screen in the given frame.

4.13.2 Event Handling Methods

All methods (except Error()) retrieve data from the widgets on the current form and send a message to the OAM Server then wait for a response. If this response contains an error, then the error code is extracted from the response and an error event is generated (i.e. call the Error procedure passing it the extracted error code). For details about the format of the messages sent to and from the OAM Server, see the External Interfaces section.

4.13.2.1 ActivatePlan()

Declaration

```
proc AcivatePlan { }
```

Purpose

Activates a plan (i.e. makes it valid for a subscription specify this plan).

Implementation Notes/Pseudocode

Send a <ActivatePlanRequest> message then wait for a <ActivatePlanResponse>.

4.13.2.2 AddCharge()

Declaration

```
proc AddCharge { }
```

Purpose

Adds a charge to a subscription.

Implementation Notes/Pseudocode

Send a <AddChargeRequest> message then wait for a <AddChargeResponse>.

4.13.2.3 AddCustomer()

Declaration

```
proc AddCustomer { }
```

Purpose

Add a customer to the OAM system.

Implementation Notes/Pseudocode

Send an <AddCustomerRequest> message then wait for a <AddCustomerResponse>.

4.13.2.4 AddEquipment() and RemoveEquipment()

Declaration

```
proc AddEquipment { }
and
proc RemoveEquipment { }
```

Purpose

Marks a slot in specified exchange and shelf as containing (Add) or not containing (Remove) a piece of equipment.

Implementation Notes/Pseudocode

Send a <AddEquipmentRequest> or <RemoveEquipmentRequest> message then wait for a <AddEquipmentResponse> or <RemoveEquipmentResponse>.

4.13.2.5 AddPlan()

Declaration

```
proc AddPlan { }
```

Purpose

Adds a plan to the OAM system.

Implementation Notes/Pseudocode

Send a <AddPlanRequest> message then wait for a <AddPlanResponse>.

4.13.2.6 AddSubscription()

Declaration

```
proc AddSubscription { }
```

Purpose

Add a subscription to the OAM System.

Implementation Notes/Pseudocode

Send an <AddSubscriptionRequest> message then wait for a <AddSubscriptionResponse>.

4.13.2.7 AssociateExchange() and DisassociateExchange()

Declaration

```
proc AssociateExchange { }
and
proc DisassociateExchagne { }
```

Purpose

Allocates (associate) or Deallocates (disassociates) a trunk card linking a local exchange to a remote exchange. This link allows the phones in the local exchange to call phones in the remote exchange (Note: this link is unidirectional).

Implementation Notes/Pseudocode

Send a <AssociateExchangeRequest> or <DisassociateExchangeResponse> message then wait for a <AssociateExchangeResponse> or <DisassociateExchangeResponse>.

4.13.2.8 CancelSubscription()

Declaration

```
proc CancelSubscription { }
```

Purpose

Cancels a subscription.

Implementation Notes/Pseudocode

Send a <CancelSubscriptionRequest> message then wait for a <CancelSubscriptionResponse>.

4.13.2.9 DeactivatePlan()

Declaration

```
proc DeactivatePlan { }
```

Purpose

Deactivates a plan. A replacement plan is also provided, all subscriptions that refer to the deactivated plan will be switched to the replacement plan.

Implementation Notes/Pseudocode

Send a <DescribePlanRequest> message then wait for a <DescribePlanResponse>.

4.13.2.10 DescribeBill()

Declaration

```
proc DescribeBill { }
```

Purpose

Retrieves all information pertaining to a bill (used, for example, when viewing a customer’s bill).

Implementation Notes/Pseudocode

Send a <DescribeBillRequest> message then wait for a <DescribeBillResponse>.

4.13.2.11 DescribeEquipment()

Declaration

```
proc DescribeEquipment { }
```

Purpose

Retrieves information about one or more pieces of equipment.

Implementation Notes/Pseudocode

Send a <DescribeEquipmentRequest> message then wait for a <DescribeEquipmentResponse>.

4.13.2.12 DescribeExchange ()

Declaration

```
proc DescribeExchange { }
```

Purpose

Retrieves information about one or more exchanges.

Implementation Notes/Pseudocode

Send a <DescribeExchangeRequest> message then wait for a <DescribeExchangeResponse>.

4.13.2.13 DescribePlan()

Declaration

```
proc DescribePlan { }
```

Purpose

Retrieves one of more plans.

Implementation Notes/Pseudocode

Send a <DescribePlanRequest> message then wait for a <DescribePlanResponse>.

4.13.2.14 EditCustomer()

Declaration

```
proc EditCustomer { }
```

Purpose

Updates a customer’s information.

Implementation Notes/Pseudocode

Send an <EditCustomerRequest> message then wait for a <EditCustomerResponse>.

4.13.2.15 EditExchange ()

Declaration

```
proc EditExchange { }
```

Purpose

Updates an exchange’s information.

Implementation Notes/Pseudocode

Send a <EditExchangeRequest> message then wait for a <EditExchangeResponse>.

4.13.2.16 EditSubscription()

Declaration

```
proc EditSubscription { }
```

Purpose

Updates a subscription’s information.

Implementation Notes/Pseudocode

Send an <EditSubscriptionRequest> message then wait for a <EditSubscriptionResponse>.

4.13.2.17 EnableEquipment() and DisableEquipment()

Declaration

```
proc EnableEquipment { }
and
proc DisableEquipment { }
```

Purpose

Marks a piece of equipment as offline-for-maintenance (Disable) or not (Enable).

Implementation Notes/Pseudocode

Send a <EnableEquipmentRequest> or <DisableEquipmentRequest> message then wait for a <EnableEquipmentResponse> or <DisableEquipmentResponse>.

4.13.2.18 Error()

Declaration

```
proc Error { errorCode }
```

Purpose

Format and display an error message

Arguments

errorCode – an number representing a specific error.

Implementation Notes/Pseudocode

Translate errorCode to a message then display that message using the DisplayError procedure.

4.13.2.19 FindCustomer()

Declaration

```
proc FindCustomer { }
```

Purpose

Retrieves one or more customers whose information matches the criteria specified on the Find Customer form.

Implementation Notes/Pseudocode

Send a <FindCustomerRequest> message then wait for a <FindCustomerResponse>.

4.13.2.20 FindSubscription()

Declaration

```
proc EditSubscription { }
```

Purpose

Retrieves one or more subscriptions whose information matches the criteria specified on the current form.

Implementation Notes/Pseudocode

Send a <FindSubscriptionRequest> message then wait for a <FindSubscriptionResponse>.

4.13.2.21 Login()

Declaration

```
Proc Login { }
```

Purpose

Log a user into the OAM system

Implementation Notes/Pseudocode

Connect to the OAM Server using the connection login method.

4.13.2.22 OutputBill()

Declaration

```
proc OutputBill { }
```

Purpose

Output a bill (used, for example, when reprinting a customer’s bill).

Implementation Notes/Pseudocode

Send a <OutputBillRequest> message then wait for a <OutputBillResponse>.

4.13.2.23 QueryCU()

Declaration

```
proc QueryCU { }
```

Purpose

Retrieves information about a piece of equipment from the CU (instead of the OAM database).

Implementation Notes/Pseudocode

Send a <QueryCURequest> message then wait for a <QueryCUResponse>.

4.13.2.24 ResetEquipment()

Declaration

```
proc ResetEquipment { }
```

Purpose

Resets a piece of equipment (returns the equipment to an idle state regardless of its current state).

Implementation Notes/Pseudocode

Send a <ResetEquipmentRequest> message then wait for a <ResetEquipmentResponse>.

4.13.2.25 ResumeSubscription()

Declaration

```
Proc ResumeSubscription { }
```

Purpose

Resumes a subscription.

Implementation Notes/Pseudocode

Send a <ResumeSubscriptionRequest> message then wait for a <ResumeSubscriptionResponse>.

4.13.2.26 SuspendSubscription()

Declaration

```
proc SuspendSubscription { }
```

Purpose

Suspends a subscription.

Implementation Notes/Pseudocode

Send a <SuspendSubscriptionRequest> message then wait for a <SuspendSubscriptionResponse>.

4.13.2.27 TestEquipment()

Declaration

```
proc TestEquipment { }
```

Purpose

Tests a piece of equipment.

Implementation Notes/Pseudocode

Send a <TestEquipmentRequest> message then wait for a <TestEquipmentResponse>.

4.13.3 Connection Methods

4.13.3.1 Close()

Declaration

```
proc Close(}
```

Purpose

Close the previously connected socket and unbind the previously bound socket.

4.13.3.2 Login()

Declaration

```
proc Login { srv port userID password }
```

Purpose

Connects to a given server with the user name and password specified. If successful, it binds to a listening socket to accept incoming communications.

Arguments

Srv, port – the hostname and port of the OAM server to connect to.
UserID, password – authentication information needed to log into the OAM system.

Implementation Notes/Pseudocode

Connect to the OAM server and send a <LoginRequest> message, then wait for a <LoginResponse>. If login is successful, bind a socket to listen on port 8725. Unsolicited messages (e.g. errors) will be written to this port. If the login fails extract the error code from <LoginResponse> and generate an error event (i.e. call the Error procedure, passing it the error code)

Implementation Notes/Pseudocode

Serialize the DOM Node passed in into a string and write the string on a previously connected socket.

4.13.3.3 Recv()

Declaration

```
proc Recv(}
```

Purpose

Returns a DOM Node corresponding to the message received.

Implementation Notes/Pseudocode

Receive on the previously connected socket and then deserialize the string into a DOM Node and return it.

4.13.3.4 Send()

Declaration

```
proc Send {top}
```

Purpose

Sends a message out on a previously established connection.

Arguments

top – a variable that is referencing a XML DOM Node that contains the message to be sent.

5.0 External Interfaces

5.1 Operator Interface

The Operator Interface is specified in the SRS. The User Interface has been modified with changes focused on form organization. Screen captures of the interface to be implemented are presented in Appendix B. The new UI will be easier to navigate and implement over the one proposed in the initial SRS because it takes advantage of the similarities between certain forms (e.g. Add and Edit Customer). Refer to the SRS for a complete specification of the functionality of each widget; note however that some of the widgets have been relocated.

5.2 Message format

The Message format between the OAM Server and the Client uses a payload of XML Data. It is written in 8-bit ASCII, lines are separated by CRLF pairs. The End Transmission of single message is determined by sending two subsequent CRLF pairs. The Document Type Definition (DTD) for the XML data is specified below. An example of a request (with CRLF pairs) would be:

```
<Message><LoginRequest>\r\n
<UserID>dftapuska</UserID><Password>password</Password>\r\n
</LoginRequest></Message>\r\n\r\n
```

The DTD allows quick validation performed by the XML parsing engine rather than code that must be written into the Server to verify that all the elements are in the right order. There are quite a few messages in the DTD, each Message Type corresponds to two messages; a request and a response. The 2*n messages allows the DTD to be strongly typed, so that it is location of data is guaranteed, so the application may be specific about assume the location and quantity of data received. An initial draft

```
<?xml version="1.0"?>
<!DOCTYPE Message [
  <!-- Data Types -->
  <!-- Login / Authentication Data -->
  <!ELEMENT UserId (Letter+)>
  <!ELEMENT Password ((Letter|Digit)+)>

  <!ELEMENT SubInfo (SubscriptionID, CustomerID, CheckSum, Address,
           City, Province, PostalCode, Status, COS?, DN?, EX?,
           Shelf?, Slot?, CallPlan?)>
  <!ELEMENT CustInfo (CustomerID, CheckSum, FirstName, LastName,
            Email, Address, City, Province, PostalCode, SubInfo+ )>
  <!ELEMENT Charge (ChargeAmount, ChargeType, StartDate, EndDate)>
  <!ELEMENT Call (StartDate, StartTime, Duration, PhoneNumberCalled,
           ChargeAmount, CallPlan, IsLocal)>
  <!ELEMENT BillInfo (CustInfo, StartDate, EndDate, Total, Charge+,
           Call+ )>
  <!ELEMENT Rate (EX, CheckSum, ChargeRate)>
  <!ELEMENT ExchInfo (EX, CheckSum, Location, AssociatedEX+, Rate+)>
  <!ELEMENT CallPlan (CallPlanID, CallPlanName, Valid, ChargeRate)>
  <!ELEMENT Equipment (EX, Shelf, Slot, Checksum, DN?, COS?, STAT?)>
  <!ELEMENT SubCriteria (SuscriptionID?, CustomerID?, EX?, Address?,
           City?, Province?, PostalCode?, DN?, COS?, CallPlan?)>
  <!ELEMENT SubChange   (SuscriptionID?, CustomerID?, EX?, Address?,
           City?, Province?, PostalCode?, DN?, COS?, CallPlan?)>

  <!-- Customer Data -->
  <!ELEMENT FirstName (Letter+)>
  <!ELEMENT LastName (Letter+)>
  <!ELEMENT Email (#PCDATA+)>
  <!ELEMENT Address (#PCDATA+)>
  <!ELEMENT City (#PCDATA+)>
  <!ELEMENT Province (#PCDATA+)>
  <!ELEMENT PostalCode (#PCDATA+)>
  <!ELEMENT CustomerID (Letter|Digit)+>

  <!-- Subscription Data -->
  <!ELEMENT SubscriptionID (Letter|Digit)+>
  <!ELEMENT Status Digit>

  <!-- Billing Data -->
  <!ELEMENT ChargeAmount (#PCDATA)>
  <!ELEMENT ChargeType Digit>
  <!ELEMENT IsLocal Digit>
  <!ELEMENT Total (#PCDATA)>

  <!-- Call Plan Data -->
  <!ELEMENT CallPlanID Digit+>
  <!ELEMENT ReplacementPlanID Digit+>
  <!ELEMENT CallPlanName #PCDATA+>
  <!ELEMENT ChargeRate Digit+>
  <!ELEMENT Valid Digit>

  <!-- CU Data -->
  <!ELEMENT Shelf Digit+>
  <!ELEMENT Slot Digit+>
  <!ELEMENT DN (Digit)+>
  <!ELEMENT EX Digit+>
  <!ELEMENT AssociatedEX Digit+>
  <!ELEMENT COS Digit+>
  <!ELEMENT STAT Digit+>
  <!ELEMENT TrunkNo Digit+>

  <!-- Misc Data -->
  <!ELEMENT CheckSum (Digit)+>
  <!ELEMENT Error Digit+)
  <!ELEMENT StartDate (#PCDATA)>
  <!ELEMENT EndDate (#PCDATA)>
  <!ELEMENT StartTime (#PCDATA)>
  <!ELEMENT Duration (Digit+)>
  <!ELEMENT PhoneNumberCalled (Digit+)>

  <!-- Request Types -->
  <!ELEMENT LoginRequest (UserId, Password)>
  <!ELEMENT LoginResponse (UserId, Error?)>
```

```
<!-- Customer Ops -->
<!ELEMENT AddCustomerRequest (FirstName, LastName, Email, Address,
         City, Province, PostalCode)>
<!ELEMENT AddCustomerResponse (CustInfo, Error?)>
<!ELEMENT FindCustomerRequest (CustomerID?, FirstName?, LastName?,
         Email?, Address?, City?, Province?, PostalCode?, EX?,
         DN?)>
<!ELEMENT FindCustomerResponse (CustInfo+, Error?)>
<!ELEMENT EditCustomerRequest (CustomerID, CheckSum, FirstName?,
         LastName?, Email?, Address?, City?, Province?,
         PostalCode?)>
<!ELEMENT EditCustomerResponse (CustInfo, Error?)>

<!-- Subscription Ops -->
<!ELEMENT AddSubscriptionRequest (CustomerID, EX, Address, City,
         Province, PostalCode, DN?, COS, CallPlan?)>
<!ELEMENT AddSubscriptionResponse (SubInfo, Error?)>
<!ELEMENT CancelSubscriptionRequest SubscriptionID, CustomerID>
<!ELEMENT CancelSubscriptionResponse (SubInfo, Error?)>
<!ELEMENT FindSubscriptionRequest (CustomerID?, SubscriptionID?,
         EX?, DN?, Address?, City?, Province?, PostalCode?, COS?,
         CallPlan?)>
<!ELEMENT FindSubscriptionResponse (SubInfo+, Error?)>
<!ELEMENT EditSubscriptionRequest (SuscriptionID, CheckSum,
         CustomerID, EX?, Address?, City?, Province?, PostalCode?,
         DN?, COS?, CallPlan?)>
<!ELEMENT EditSubscriptionResponse (SubInfo, Error?)>
<!ELEMENT SuspendSubscriptionRequest SubscriptionID, CustomerID>
<!ELEMENT SuspendSubscriptionResponse (SubInfo, Error?)>
<!ELEMENT ResumeSubscriptionRequest SubscriptionID, CustomerID>
<!ELEMENT ResumeSubscriptionResponse (SubInfo, Error?)>
<!ELEMENT EditAllSubscriptionsRequest SubCriteria, SubChange>
<!ELEMENT EditAllSubscriptionsResponse Error?>

<!-- Billing Ops -->
<!ELEMENT OutputBillRequest SubscriptionID, CustomerID, StartDate,
         EndDate>
<!ELEMENT OutputBillResponse Error? >
<!ELEMENT OutputAllBillsRequest EX? >
<!ELEMENT OutputAllBillsResponse Error? >
<!ELEMENT DescribeBillRequest SubscriptionID, CustomerID,
         StartDate, EndDate>
<!ELEMENT DescribeBillResponse (BillInfo, Error?)>

<!-- Charges and Calls -->
<!ELEMENT AddChargeRequest SubscriptionID, CustomerID,
         ChargeAmount, ChargeType, StartDate, EndDate>
<!ELEMENT AddChargeResponse Error?>
<!ELEMENT BillCallRequest StartDate, StartTime, Duration, EX,
         Shelf, Slot, PhoneNumberCalled>
<!ELEMENT BillCallResponse Error?>
<!ELEMENT AddChargeAllRequest EX?, ChargeAmount, ChargeType,
         StartDate, EndDate>
<!ELEMENT AddChargeAllResponse Error?>

<!-- CallPlan Stuff -->
```

```
  <!ELEMENT AddPlanRequest CallPlanName, ChargeRate>
  <!ELEMENT AddPlanResponse (CallPlan, Error?)>
  <!ELEMENT DescribePlanRequest CallPlanID?>
  <!ELEMENT DescribePlanResponse (CallPlan+, Error?)>
  <!ELEMENT ActivatePlanRequest CallPlanID>
  <!ELEMENT ActivatePlanResponse Error?>
  <!ELEMENT DeactivatePlanRequest CallPlanID, ReplacementPlanID>
  <!ELEMENT DeactivatePlanResponse Error>

  <!-- Equipment Stuff -->
  <!ELEMENT AddEquipmentRequest (EX, Shelf, Slot, Checksum)>
  <!ELEMENT AddEquipmentResponse (Ex, Shelf, Slot, Error?)>
  <!ELEMENT RemoveEquipmentRequest (EX, Shelf, Slot, Checksum)>
  <!ELEMENT RemoveEquipmentResponse (EX, Shelf, Slot, Error?)>
  <!ELEMENT TestEquipmentRequest (EX, Shelf, Slot)>
  <!ELEMENT TestEquipmentResponse (EX, Shelf, Slot, Error?)>
  <!ELEMENT ResetEquipmentRequest (EX, Shelf, Slot)>
  <!ELEMENT ResetEquipmentResponse (EX, Shelf, Slot, Error?)>
  <!ELEMENT EnableEquipmentRequest (EX, Shelf, Slot, Checksum)>
  <!ELEMENT EnableEquipmentResponse (EX, Shelf, Slot, Error?)>
  <!ELEMENT DisableEquipmentRequest (EX, Shelf, Slot, Checksum)>
  <!ELEMENT DisableEquipmentResponse (EX, Shelf, Slot, Error?)>

  <!ELEMENT DescribeEquipmentRequest (EX, Shelf?, Slot?)>
  <!ELEMENT DescribeEquipmentResponse (Equipment+, Error?)>
  <!ELEMENT QueryCURequest (Shelf, Slot, EX)>
  <!ELEMENT QueryCUResponse (Shelf, Slot, EX, (DN, COS, STAT)?,
           Error?)>

  <!-- Exchange Stuff -->
  <!ELEMENT DescribeExchangeRequest EX?>
  <!ELEMENT DescribeExchangeResponse ExchInfo+>
  <!ELEMENT EditExchangeRequest (EX, CheckSum, ChargeRate?,
           Location?, Rate+?)>
  <!ELEMENT EditExchangeResponse ExchInfo>
  <!ELEMENT AssociateExchangeRequest (EX, Shelf, Slot, CheckSum,
           RemoteEX)>
  <!ELEMENT AssociateExchangeResponse Error?>
  <!ELEMENT DisassociateExchangeRequest (EX, Shelf, Slot, CheckSum)>
  <!ELEMENT DisassociateExchangeResponse Error?>
]>
```

5.3 Database

The communication with the database will use a well define MySQL++ API. This API allows SQL queries to be written in C++ and executed on a remote database machine. The protocol is transparent to the application.

The Database Schema is an external entity and is shown below in the following Schema Diagram. Appendix C shows the DML for the Database Schema shown.

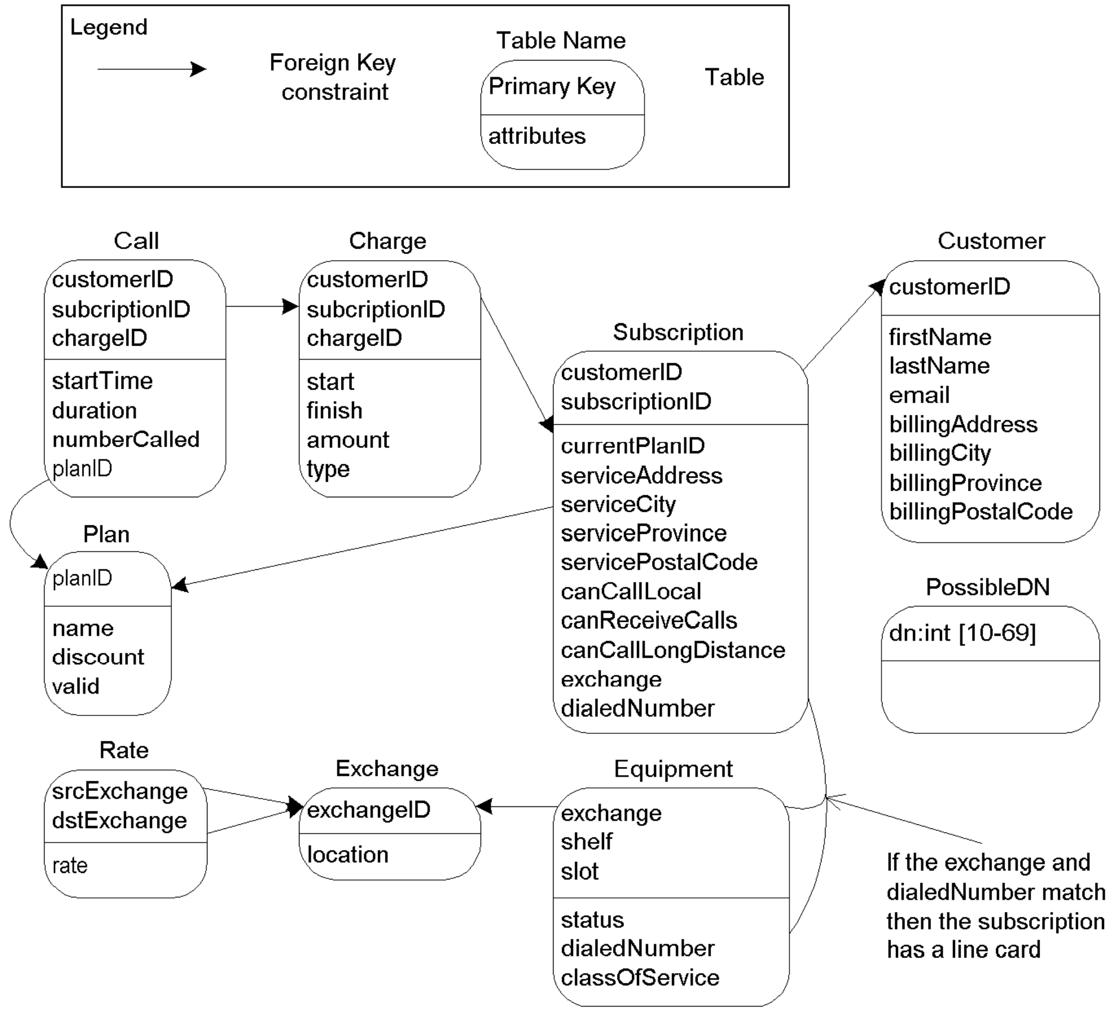


Figure 2 Database Schema Diagram

5.3.1 Notes

- PossibleDN contains one entry of each possible dialed number (10-69). It is used to improve the efficiency of finding a free phone number (see CreateSubscription() for details on how this table is used). The alternative would be to return all dialed number from the Subscription table then scan for a gap (this would be extremely inefficient if the range of dialed number were ever increased).
- Since we do not maintain historic information about a subscription's plan the Call table must store the plan id that applies to the call.
- Plans cannot be edited or deleted, because a customer may ask the Operator to reprint an old bill (this would require recalculating the old bill, and hence may require information about the old plan)
- Charges.type is one of 0=LocalCall, 1=LDCall, 2=Payment, 3=Credit, 4=ServiceFee

- There should be a dependancy between Subscription.serviceAddress (the location of the phone) and Exchange.location. However, the design team has chosen to ignore this constraint since it is not a requirement.
- An exchange's local rate is stored in the Rate table as an entry with srcExchange=dstExchange
- The exchange, dialednumber, and class of service related attributes must be stored in the Subscription table and in the Equipment table to meet the requirements. That is, to ‘reserve’ a subscription’s dialed number (e.g. when a subscription is suspended) and to maintain the class of service information on a subscription these fields must be stored in the Subscription table. Whereas, maintaining a master copy of the exchange database requires that these fields are recorded in the Equipment table.
- A subcription has a line card if the following query returns a row:
  SELECT * FROM Subscription as S, Equipment as E
  WHERE S.exchange=E.exchange AND S.dialedNumber=E.dialedNumber

5.4 Output Server

Outputting of bills requires two operations, email and printing. For simplicity, we make use of three Approved Request For Comments that specify the protocols which we are going to communicate on to do these two operations. Firstly, there is an assumption that there is Line Printer Daemon (as specified in RFC 1179) that is capable of receiving requests to print postscript files. Secondly, there is a SMTP server available to receive requests to send email [See RFC 821]. The emailing of bills will consist of emails being sent via the SMTP server. The data of the email will consist of a base 64 encoded attachment PDF file. The files will be attached according to the MIME RFC 1341.

5.5 Configuration Files

Configuration files will be required to start the dispatcher and let it know where the individual CUs are located. Individual CU Handlers do not need any configuration files because the arguments specified when you start the Handler permit the Handler to connect on different ports. The configuration file for the dispatcher will have various members such as database host, database port, CU Handler Hosts and CU Handler Ports. Whenever this configuration file is modified, the server will need to be restarted. This is an unfortunate side effect, however it could be enhanced that when the file is changed a program notifies the dispatcher that the file has changed. This is beyond what we wish to implement, but it shows that the system is extendible. An example of the configuration file is shown below:

```
#this is a comment, specify the db server by host:port
DbServer=db.uwaterloo.ca:3123
#
# for CUs specify host:port of CUHandler
# exchange number is handled by CUi where i is the exchange
# number
```

```
CU1=cu.uwaterloo.ca:8724
CU2=cu.uwaterloo.ca:8725
CU3=cu.uwaterloo.ca:8726
```

6.0 Integration Task Plan

Since our system is composed of small atomic components and there are sufficient time constraints, the Integration Task Plan should have a test plan that is flexible and not overwhelming. We feel that the system should use a Bottom-Up Integration style, meaning that small individual components will be integrated and tested until the whole system is composed. We have designed into our Task Scheduling four major integration milestones; those are Connection Worker Integration, Client Integration, Server Integration and OAM System Integration. There are three other minor integration milestones, however these have been rolled in with some coding tasks at the specific task points; these tasks are O&A Complex Ops, Maintenance Complex Ops, and Billing Complex Ops.

Although integration will occur on smaller components, for example Billing Complex Operations will need to integrate with Print Simple Ops, CU Simple Ops and Database Simple Ops, it is felt that integration on these components will occur with a Big Bank approach. The components below these three components are ensured to be unit tested, and therefore the integration of these units should not lead to a long time spent trying to find errors since the components at that level are relatively small.

The major milestones presented will occur with the Bottom-Up Integration style where drivers are written for each and the components will be tested. The detailed design allows for efficient testing of these components since message passing is done via XML. This means that a single driver could be written so that it could be reused in integration testing of various components that communicated via XML. For example If we were to test the Connection Worker or Client, we could have a single driver that played back a series of XML commands to each and test the responses against expected responses. A benefit of this is that is can be automated, so regression testing in the maintenance cycle is trivial.

Integration Task	Client	O&A	Maintenance	Billing	Connection Worker	Server	System
Client Comm.	X						X
Client Event Handler	X						X
UI Forms	X						X
Database Schema		X	X	X	X	X	X
DB Simple Ops		X	X	X	X	X	X
CU Handler		X	X	X	X	X	X
CU Simple Ops		X	X	X	X	X	X
Print Simple Ops				X	X	X	X
O&A C.O.		X			X	X	X
Maintenance C.O.			X		X	X	X
Billing C.O				X	X	X	X

Connection Worker Process					X	X	X
Billing Daemon							X
Dispatcher Process						X	X

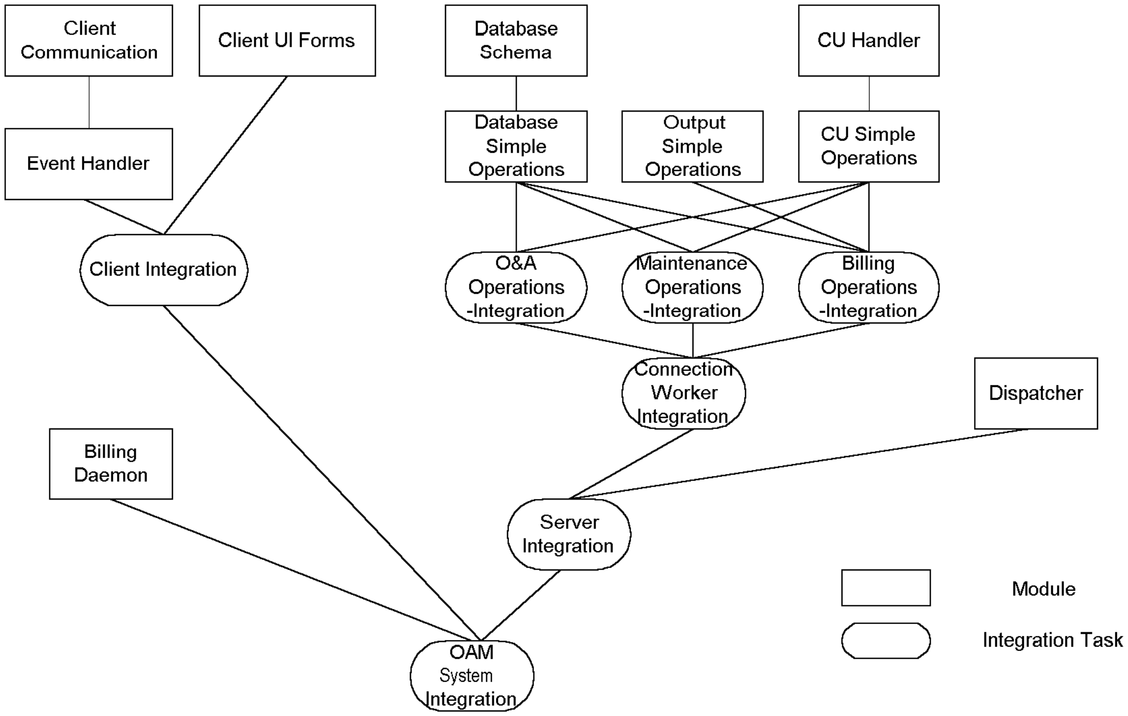


Figure 3 Integration Task Diagram

7.0 Task Scheduling

7.1 Cost Estimates

We estimate the amount of code to be roughly 12,000 lines. This estimate was calculated from the following matrix. Note the duration days column does not include integration and testing of the modules. It is a representation only for the implementation of the individual modules. The schedule provided later takes into account integration of the components.

Module	Lines of Code	Duration Days
Client Communication	200	1
Client Event Handler	2000	2
UI Forms	3000	7
Database Simple Operations	1000	2
Print Simple Operations	250	1
CU Simple Operations	250	1
CU Handler	1500	1
O&A Complex Operations	1000	2
Maintenance Complex Operations	500	1
Billing Complex Operations	500	1
Connection Worker Process	500	2
Dispatcher Process	500	2
Billing Daemon	300	1
Total	11,500	24

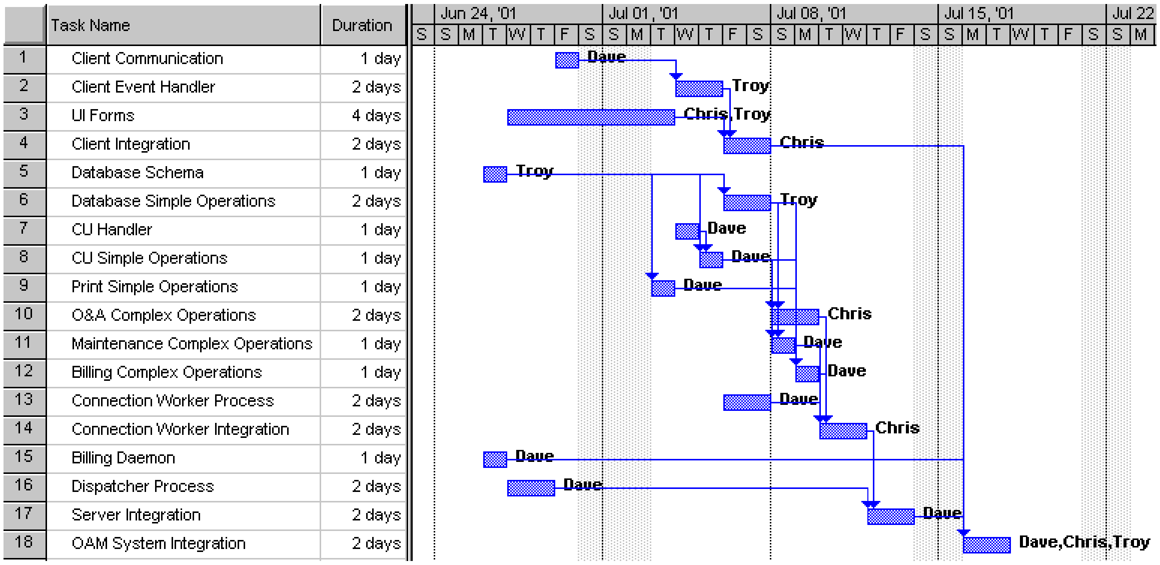
Applying COCOMO 1 on this (http://www.jsc.nasa.gov/bu2/COCOMO.html), with Development having 12 KDSI, and $3200/PM and Maintenance having 2 KDSI added annual and 1 KDSI changed annual. The following table shows the results:

Inputs	**Outputs**
Development KDSI : 12 Development Mode: Organic Average Cost Rate ($/PM): $3200 Maintenance KDSI added: 2 KDSI changed: 1 Average Cost Rate ($/PM): $3200	Effort: 33 PM Schedule: 9 months Development Cost: $105,600 Productivity: 364 instructions per month Average Staffing: 3.7 FT programmers Annual Maintenance: 8 PM Annual Maintenance Cost: $25,600

The difference between the COCOMO expected schedule and the schedule below is quite different. This was anticipated because we need to implement the system in a set amount of time (less than three weeks) so our schedule must fit into that timeline. Also the COCMO estimates include: Plans and requirements, Product Design, Detailed Design, Code and unit test, Integration and test whereas our schedule below assumes that many of these components have been completed or will not be completed. The schedule below shows Code and unit Test with Integration (and some Integration Tests), however it does not anticipate that the system will be fully tested upon coding completion.

7.2 Task Schedules

The following Gantt Chart has been provided that details the Task Scheduling of the three programming resources; Dave, Troy and Chris. The total hours assigned to each are 88 hours to Troy, 90 hours to Chris and 120 hours to Dave. Although Dave has a greater amount of hours committed, it is believe that his schedule will progress rapidly since the tasks he has are relatively small. The likelihood for delay on the small tasks is minimal, so we feel that Dave's schedule should not be delayed. We have planned for a long weekend (June 30-July 2$^{nd)}$ in the schedule to accommodate vacation time. This still allows at least one week of possible delay in the schedule and we will not be delayed for the project deadline, nor any components missing.



	Task Name	Duration
1	Client Communication	1 day
2	Client Event Handler	2 days
3	UI Forms	4 days
4	Client Integration	2 days
5	Database Schema	1 day
6	Database Simple Operations	2 days
7	CU Handler	1 day
8	CU Simple Operations	1 day
9	Print Simple Operations	1 day
10	O&A Complex Operations	2 days
11	Maintenance Complex Operations	1 day
12	Billing Complex Operations	1 day
13	Connection Worker Process	2 days
14	Connection Worker Integration	2 days
15	Billing Daemon	1 day
16	Dispatcher Process	2 days
17	Server Integration	2 days
18	OAM System Integration	2 days

8.0 Programming Conventions

The Programming Style that we anticipate to be used is a modified Hungarian notation. Hungarian Notation is well defined in naming methods, and variables. A copy of the Hungarian Notation can be found at the Microsoft MSDN website. http://msdn.microsoft.com/library/techart/hunganotat.htm

Our extension of Hungarian notation is to specify that all variables will be named in the following fashion: for global variables g_variable, for member variables m_variable and for static variables s_variable. Variables inside a local scope need not be named as our convention specifies, the convention only applies to variables in the global scope. It will also be required that preprocessor definitions must always be in all capitols. For C/C++ code, C++ style comments (ie. //) must be used inside method definitions and not /* */. Tabs are not to be used in the code, and spaces used instead.

9.0 References

Apache Xerces C++ XML Library
http://xml.apache.org/xerces-c/index.html

CS 445 Project Introduction, August 2000.
http://www.student.math.uwaterloo.ca/~cs445/project/intro.pdf

CS 445 Software Interface Description, August 2000.
http://www.student.math.uwaterloo.ca/~cs445/project/soft.pdf

COCOMO I, Cost Estimates
http://www.jsc.nasa.gov/bu2/COCOMO.html

Microsoft Development Network – Coding Conventions
http://msdn.microsoft.com/library/techart/hunganotat.htm

MySQL Manual
http://www.mysql.com/Downloads/Manual/manual.pdf

Design Document Example
http://www.cs.cornell.edu/database/predator/designdoc.html

10.0 Data Dictionary

Word	Meaning
API	Application Programming Interface
COCOMO	Constructive Cost Model
CRC	Class, Responsibilities, Collaborator
CRUD	Create, Restore, Update, Delete
CU	Control Unit
CS 445 SRS	Refers to the requirements document for the OAM Software System (*CS445 Software Requirements Specification: OAM Software for SX4*)
DB	Database
DOM	Document Object Model
DN	Dialed Number
DTD	Document Type Definition
EX	Exchange
GUI	Graphical User Interface
KDSI	Kilo deliverable source instruction
MIME	Multipurpose Internet Mail Extensions
MSDN	Microsoft Development Network
O&A	Operations and Administration
OAM	Operating, Administration and Maintenance
PDF	Portable Document Format
PM	Programmer Month
RFC	Request for Comments
Spawn	The act of copying a process image and then replacing the code image inside the process image with that of another program. In Win32, this is a direct method call, in UNIX it can be simulated via a fork followed by an execl.
SMTP	Simple Mail Transfer Protocol
SRS	Software Requirements Specification
SQL	Structured Query Language
UI	User Interface
XML	Extensible Markup Language

11.0 Appendix A – Examples of XML Messages

Example of Login Request
This example illustrates a client sending login request, the Request is rejected, and an error is returned. The error returned corresponds to invalid password. (See External interfaces for error codes)

Request

```
<Message>
 <LoginRequest>
  <UserID>dftapuska</UserID>
  <Password>Dave</Password>
 </LoginRequest>
</Message>
```

Response

```
<Message>
 <LoginResponse>
  <UserID>dftapuska</UserID>
   <Error>04</Error>
 </LoginResponse>
</Message>
```

Example of AddCustomer Request
This example illustrates a client sending an add customer request. The Customer is added and the CustomerID is returned

Request

```
<Message>
 <AddCustomerRequest>
  <FirstName>Dave</FirstName>
  <LastName>Tapuska</LastName>
  <Email>d@tapuska.com</Email>
  <Address>23 Glen Dr</Address>
  <City>Waterloo</City>
  <Province>Ont</Province>
  <PostalCode>
    Z1Z2Z2
  </PostalCode>
 </AddCustomerRequest>
</Message>
```

Response

```
<Message>
 <AddCustomerResponse>
  <CustInfo>
   <CustomerID>3123</CustomerID>
   <CheckSum>1AD2DF23</CheckSum>
   <FirstName>Dave</FirstName>
   <LastName>Tapuska</LastName>
   <Email>d@tapuska.com</Email>
   <Address>23 Glen Dr</Address>
   <City>Waterloo</City>
   <Province>Ont</Province>
   <PostalCode>
        Z1Z2Z2
   </PostalCode>
   <Status>2</Status>
  </CustInfo>
 </AddCustomerResponse>
</Message>
```

Example of FindCustomer Request
This example illustrates a client sending a find customer request. Once customer that has the first name of Dave is returned. If there were more customers with that firstname, they would be returned.

Request

```
<Message>
 <FindCustomerRequest>
  <FirstName>Dave</FirstName>
 </FindCustomerRequest>
</Message>
```

Response

```
<Message>
 <FindCustomerResponse>
  <CustInfo>
   <CustomerID>3123</CustomerID>
   <CheckSum>1AD2DF23</CheckSum>
   <FirstName>Dave</FirstName>
   <LastName>Tapuska</LastName>
   <Email>d@tapuska.com</Email>
   <Address>23 Glen Dr</Address>
   <City>Waterloo</City>
   <Province>Ont</Province>
   <PostalCode>
        Z1Z2Z2
   </PostalCode>
   <Status>2</Status>
  </CustInfo>
 </FindCustomerResponse>
</Message>
```

Example of EditCustomer Request

This is an example of an Edit Customer request and response that fails. It fails because the cookie or checksum that passed along with the request is invalid or does not correspond to the one that the database stores.

Request

```
<Message>
 <EditCustomerRequest>
  <CustID>3123</CustID>
  <Checksum>12322</CheckSum>
  <FirstName>Bob</FirstName>
 </EditCustomerRequest>
</Message>
```

Response

```
<Message>
 <EditCustomerResponse>
  <CustInfo>
   <CustomerID>3123</CustomerID>
   <CheckSum>1AD2DF23</CheckSum>
   <FirstName>Dave</FirstName>
   <LastName>Tapuska</LastName>
   <Email>d@tapuska.com</Email>
   <Address>23 Glen Dr</Address>
   <City>Waterloo</City>
   <Province>Ont</Province>
   <PostalCode>
        Z1Z2Z2
   </PostalCode>
   <Status>2</Status>
  </CustInfo>
  <Error>09</Error>
 </EditCustomerResponse>
</Message>
```

Example of AddSubscription Request

This is an example of an Add Subscription request and response that is successful. It allocates a subscription that corresponds to the Customer Requested.

Request

```
<Message>
 <AddSubscriptionRequest>
  <CustID>3123</CustID>
  <EX>1</EX>
  <Address>23 Glen Dr</Address>
  <City>Waterloo</City>
  <Province>Ont</Province>
  <PostalCode>
    Z1Z2Z2
  </PostalCode>
 </AddSubscriptionRequest>
</Message>
```

Response

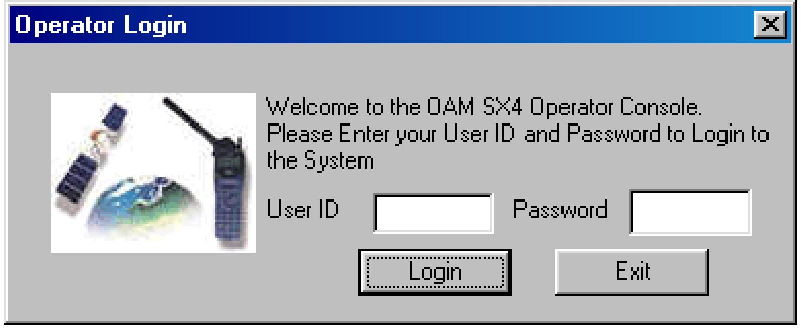
```
<Message>
 <AddSubscriptionResponse>
  <SubInfo>
   <SubscriptionID>
      2
   </SubscriptionID>
   <CustomerID>3123</CustomerID>
   <CheckSum>DEADBEEF</CheckSum>
   <Address>23 Glen Dr</Address>
   <City>Waterloo</City>
   <Province>Province</Province>
   <PostalCode>Z1Z2Z2</PostalCode>
   <Status>2</Status>
   <COS>2</COS>
   <DN>34</DN>
   <EX>1</EX>
   <Shelf>1</Shelf>
   <Slot>2</Slot>
  </SubInfo>
 </AddSubscriptionResponse>
</Message>
```

12.0 Appendix B – Enhancement of UI

It was decided that we needed to enhance the UI as the comments from the CS 445 TA said were that the UI could be more intuitive. We took these comments into account and decided to reorganize the way the controls are laid out. It is mainly a hierarchical change eliminating many screens that were repeated in the SRS. The functionality of the core screens have not changes, eg. Customer Details and Subscription Details. We have incorporated a tab control rather than the navigational "Back" control that the SRS used. We also have expressed the line cards, trunk cards in a tree control rather than a list, as the list could have hundreds of entries and would be difficult to navigate.

12.1 Operator Login

The Operator Login Control has not changed. The functionality is exactly as specified in the CS 445 SRS on Page 9.

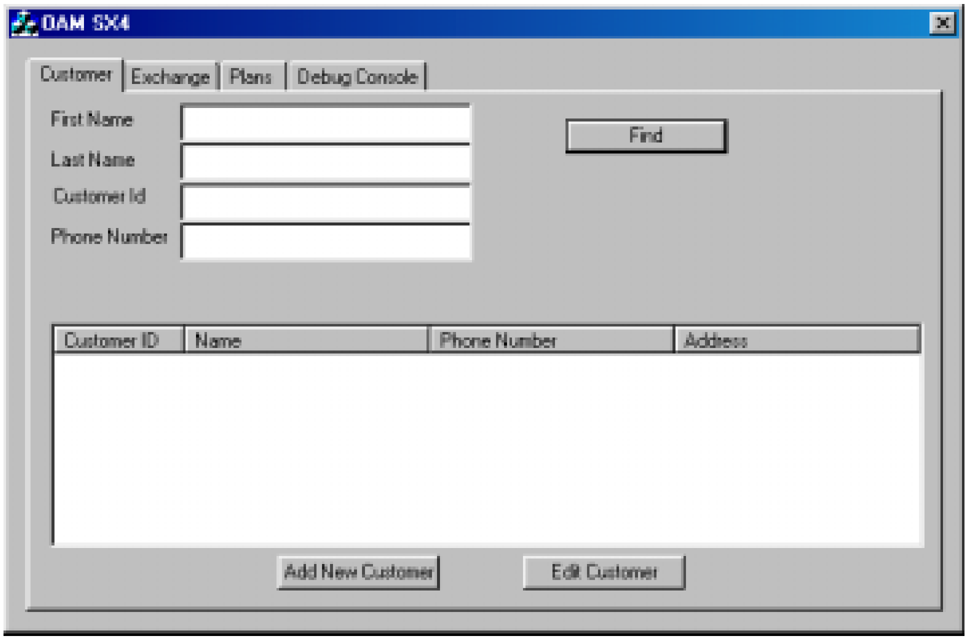


Welcome to the OAM SX4 Operator Console. Please Enter your User ID and Password to Login to the System

Figure 4 Operator Login

12.2 Main Window – Customer Operations

The Customer Operations Window has encapsulated the functionality of Customer Operations Dialog in the SRS. The use of the tab control allowed us to eliminate the need for a separate dialog. We have also placed the field fields directly on the Customer window; this is intuitive because you need to find a customer before editing it. The list of possible matches is displayed in the list control below it. Then an operator can edit the customer or add a new one.

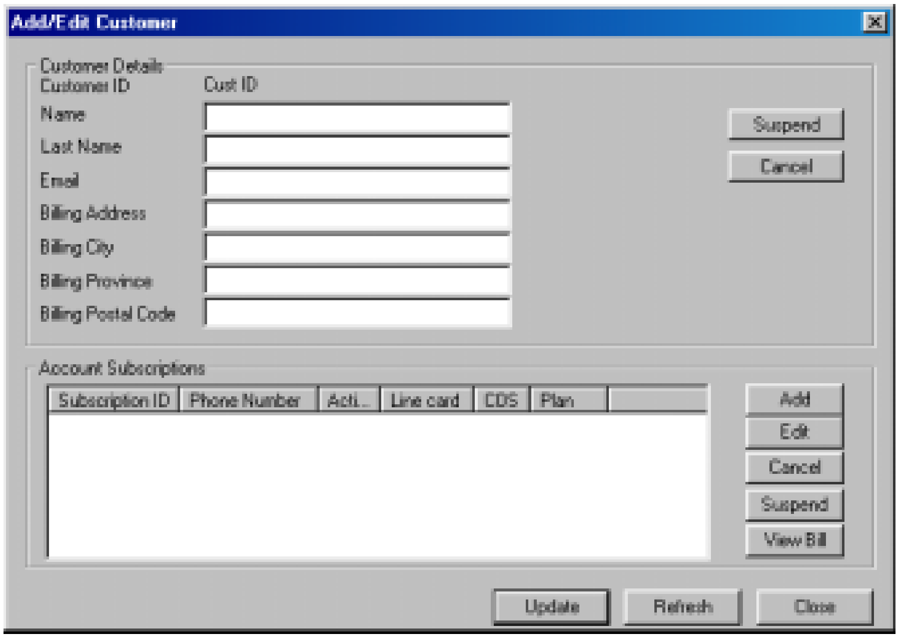


Customer ID	Name	Phone Number	Address

Figure 5 Customer Dialog

12.3 Add/Edit Customer Details

This Customer Details Form is a union of the Add, Edit, View Customer Screens in the SRS. It was felt that it is not necessary to have three separate dialogs that have pretty much the same layout and display the same information. This layout will reduce our development time as the number of dialogs has been reduced.



Subscription ID	Phone Number	Acti...	Line card	CDS	Plan

Figure 6 Add/Edit Customer

12.4 Add/Edit Subscription

The Add/Edit Subscription dialog is taken directly from the SRS; see page 30 of the SRS for details. It has only minor modifications that have the Service Address added to the Subscription and the Long Distance plan has been added to the Subscription rather than the billing details.

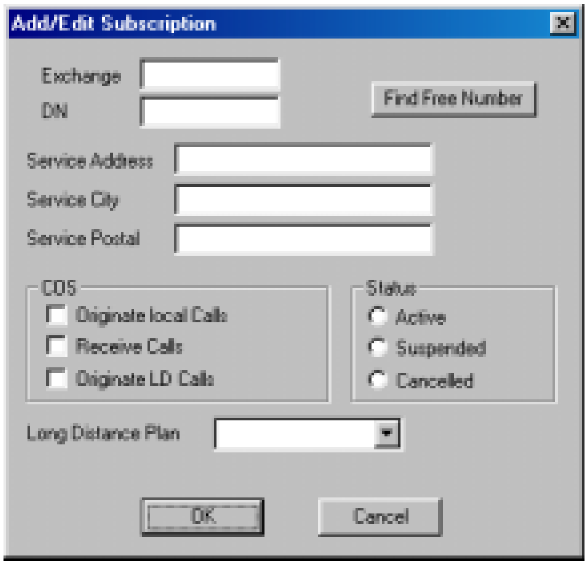


Figure 7 Add/Edit Subscription

12.5 Edit/View Bill

The Edit/View Bill Screen is very much similar to that one found in the SRS, however, it only lists charges between a certain period rather than individual bills.

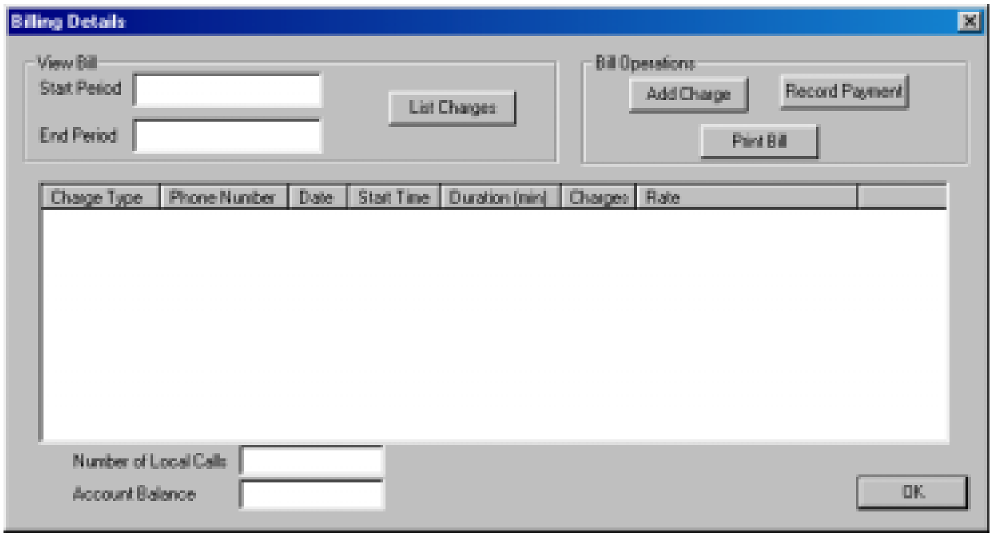


Figure 8 Billing Details

12.6 Exchange Details

The Exchange Details, Line Card Details and Trunk Shelf Screens incorporate the functionality of the Maintenance Dialog in the SRS. However, we have broken the UI up to use a Tree Control so navigation is easier.

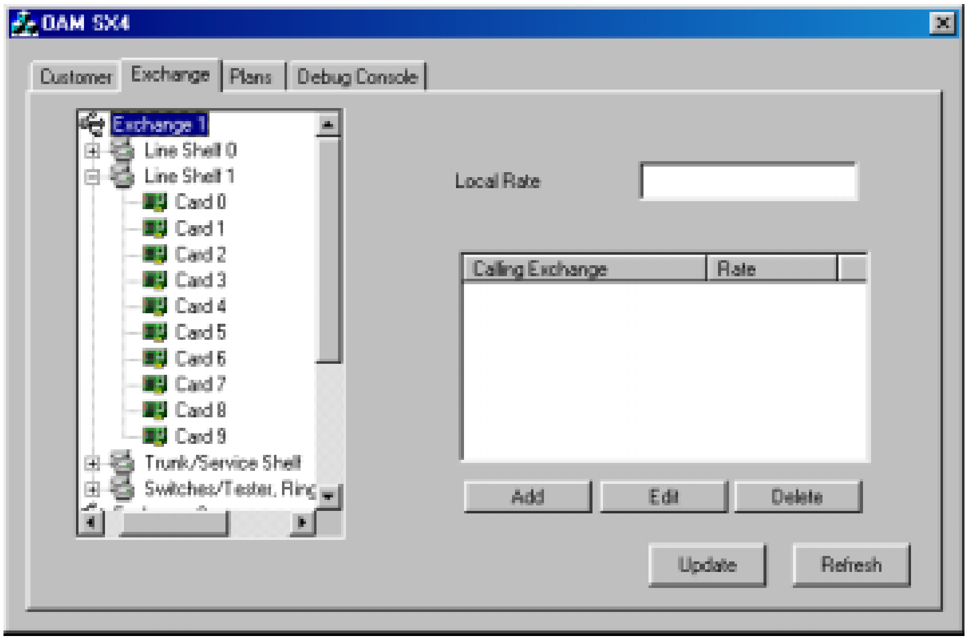


Figure 9 Exchange Details

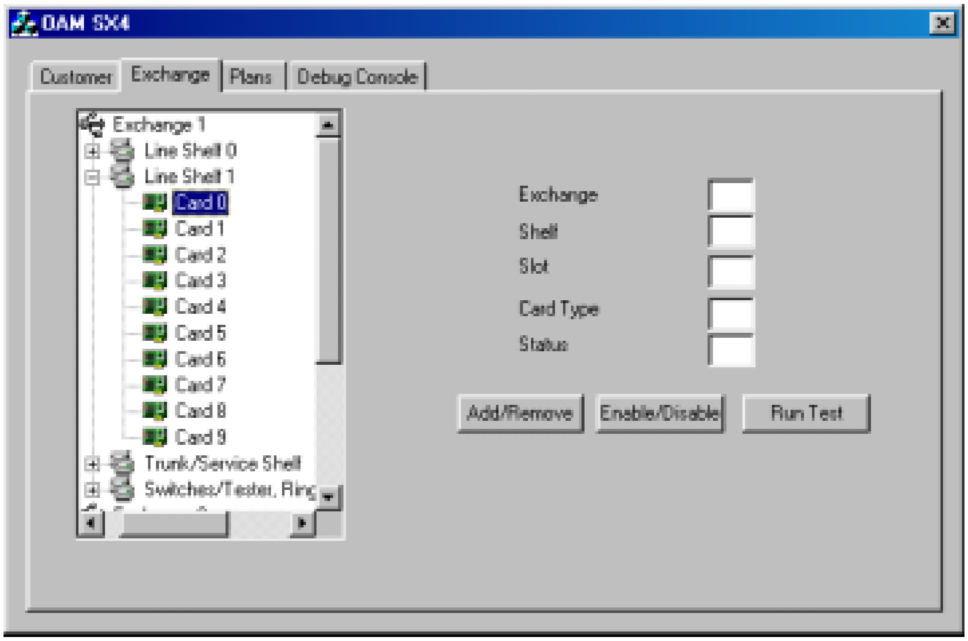


Figure 10 LineCard Details

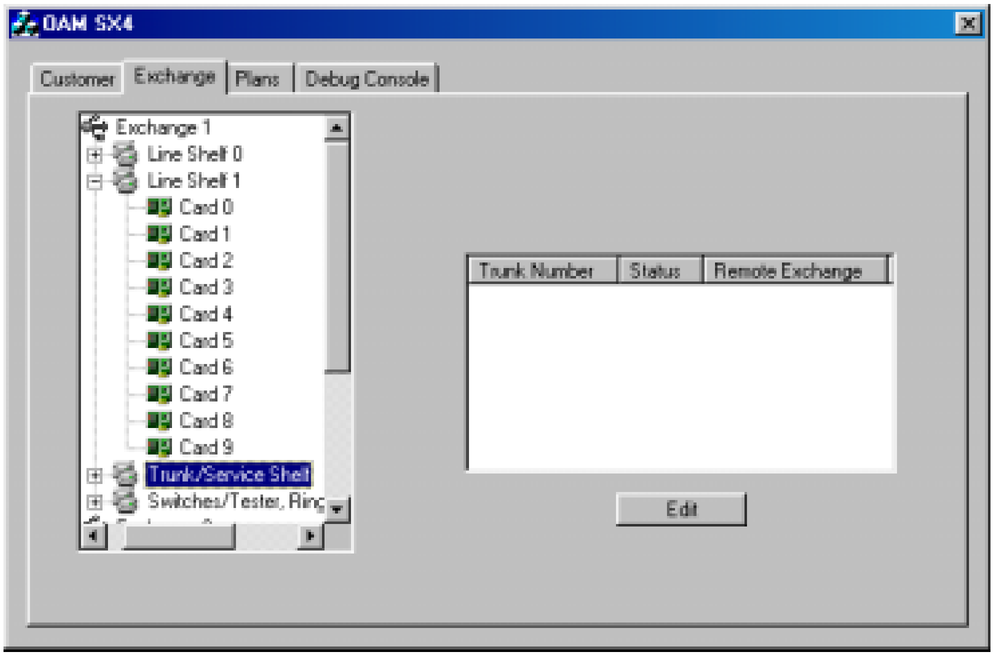


Figure 11 Trunk Card Details

12.7 Call Plan Details

The Call Plan Details and Add/Edit Call Plan Dialogs are a very simplified version of the one found in the SRS. The GUI specified in the SRS included things that were not specified in the textual SRS, this imposed an inconsistency, we have chosen to adopt the textual version and not follow the complicated UI. The following two dialogs show the functionality required by the SRS.

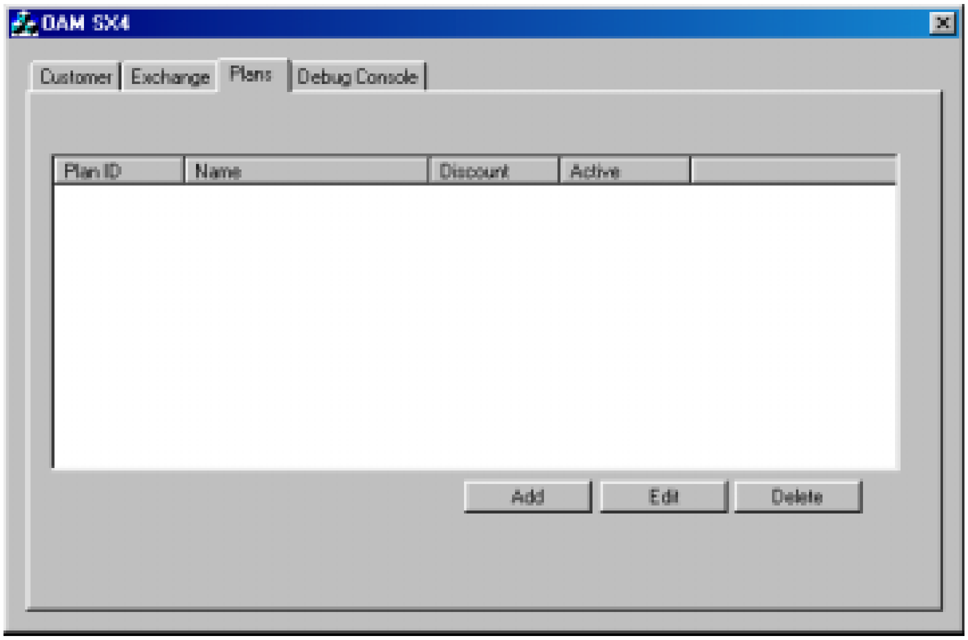


Figure 12 Call Plan Details

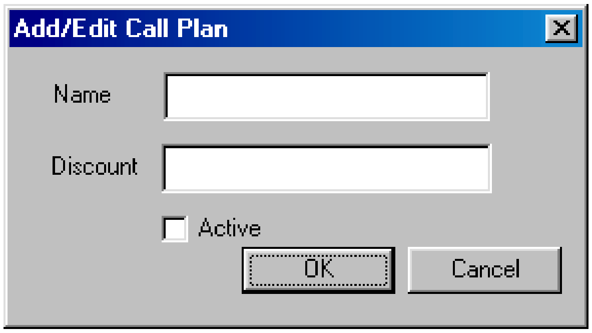
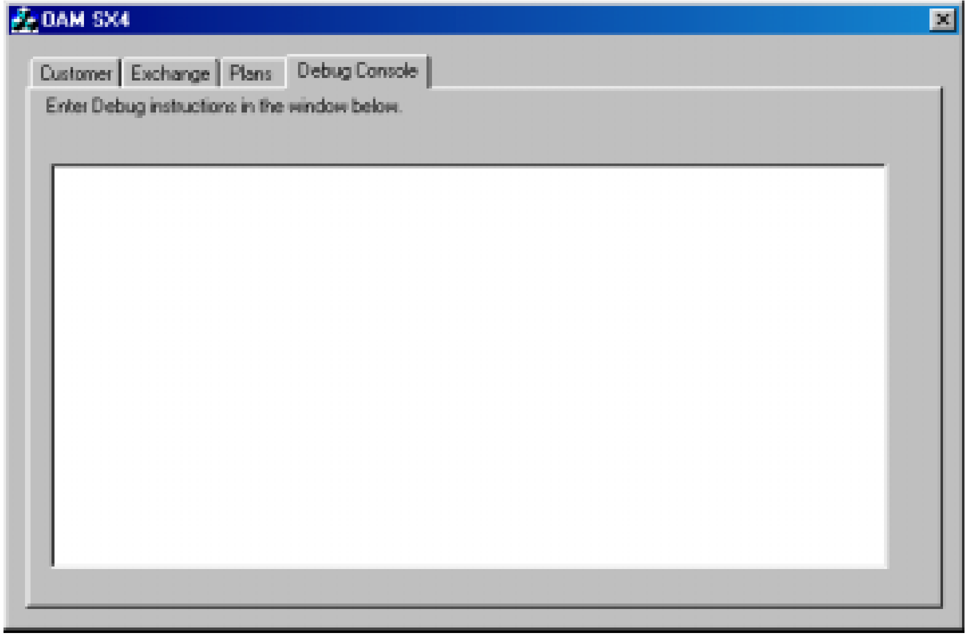


Figure 13 Add/Edit Call Plan

12.8 Debug Console

The Debug Console is much like a SQL query screen.



Enter Debug instructions in the window below.

Figure 14 Debug Console

13.0 Appendix C – Database DML

A complete schema specification on the Database is provided in the External Interfaces Section. Presented here is the Data Modeling Language for the Database; it specifies exactly what the tables and relationships are amongst the keys. This is beneficial because the Schema does not show domain constraints and restrictions.

```
create table Customer
(
    customerID            int,
    firstName             varchar(20),
    lastName              varchar(30),
    email                 varchar(50),
    billingAddress        varchar(50),
    billingCity           varchar(20),
    billingProvince       varchar(10),
    billingPostalCode     varchar(6),
    PRIMARY KEY           ( customerID )
)

create table Rate
(
    srcExchange           int,
    destExchange          int,
    rate                  decimal(5,2),
    PRIMARY KEY ( srcExchange, destExchange),
    FOREIGN KEY srcExchange REFERENCES Exchange,
    FOREIGN KEY destExchange REFERENCES Exchange
)

create table Exchange
(
    exchangeID    int,
    location      varchar(30),
    PRIMARY KEY   (exchangeID),
    CHECK         (exchangeID >= 1 AND exchangeID <= 9)
)

create table PossbileDN
(
    dialedNumber  int,
    PRIMARY KEY   (dialedNumber),
    CHECK         ( dialedNumber >= 10 AND dialedNumber <= 69)
)

create table Equipment
(
    exchange      int,
    shelf         int,
    slot          int,
    status        int,
    dialedNumber  int,
    classOfService int,
    PRIMARY KEY   (exchange, shelf, slot),
    FOREIGN KEY   (exchange) REFERENCES Exchange,
    CHECK         ( dialedNumber >= 10 AND dialedNumber <= 69),
    CHECK         ( shelf >= 0 AND shelf <= 9),
    CHECK         ( slot >= 0 AND slot <= 31)
)

create table Plan
(
    planID        int,
    name          varchar(20),
    discount      DECIMAL(1,2),
    valid         int,
    PRIMARY KEY   (planID),
    CHECK         ( valid >= 0 AND valid <= 1)
)

create table Call
(
    customerID           int,
    subscriptionID       int,
    chargeID             int,
    startTime            time,
    duration             DECIMAL(5, 2),
    numberCalled         int,
    planID               int,
    PRIMARY KEY          (customerID, subscriptionID, chargeID)
    FOREIGN KEY          (planID) REFERENCES Plan,
    FOREIGN KEY          (chargeID) REFERENCES Charge
)

create table Charge
(
    customerID           int,
    subscriptionID       int,
    chargeID             int,
    start                DATE,
    finish               DATE,
    type                 int,
    amount               DECIMAL(5, 2)
    PRIMARY KEY          (customerID, subscriptionID, chargeID),
    CHECK                ( finish >= start )
)

create table Subscription
(
    customerID           int,
    subscriptionID       int,
    currentPlanID        int,
    serviceAddress       varchar(50),
    serviceCity          varchar(20),
    serviceProvince      varchar(10),
    servicePostalCode    varchar(6),
    canRecvCalls         int,
    canCallLocal         int,
    canCallLD            int,
    exchange             int,
    dialedNumber         int,
    PRIMARY KEY   (customerID, subscriptionID),
    FOREIGN KEY   (currentPlanID) REFERENCES Plan,
    FOREIGN KEY   (exchange) REFERENCES Exchange,
    FOREIGN KEY   (customerID) REFERENCES Customer,
    CHECK         ( canCallLD >= 0 AND canCallLD <= 1),
    CHECK         ( canCallLocal >= 0 AND canCallLocal <= 1),
    CHECK         ( canRecvCalls >=0 AND canRecvCalls <= 1),
    CHECK         ( dialedNumber >= 10 AND dialedNumber <= 69)
)
```