Design and Implementation of Hardware-In-a-Loop Testing for the eft/pos 2000 System

Ruiming, Xu
(401736)

October, 2006

Project Dissertation submitted to the University of Wales Swansea
in Partial Fulfilment for the Degree of Master of Science

Department of Computer Science
University of Wales Swansea
United Kingdom

Abstract

In this dissertation, we present our design and implementation of a Hardware-In-a-Loop (HIL) testing environment for the EP2 system. EP2 stands for the eft/pos 2000. It is a joint project developed by several well-known financial companies (e.g. Diners Club Switzerland Ltd., Credit Suisse) to promote the chip-based EMV (Europay/MasterCard/Visa integrated circuit standard) payment method. Hardware-In-a-Loop (HIL) testing is a well-established testing technology to verify critical systems which particularly demand correct integration of hardware and software. This technology provides a fast and systematic method of testing which can improve the quality and efficiency of our implementation.

We first introduce the EP2 system and some necessary background knowledge about our project such as XML technology, Java programming language, testing technology, some available tools and reusable work. Then we describe our implementation in detail and demonstrate how the EP2 development tool, CEPTEST, can help to validate our implementation.

Contents

1. Introduction

In modern times, people depend heavily on various IT products which have been a necessity in our daily life over the past few years. One favorable example is electronic payment systems such as the EP2 system. EP2 stands for eft/pos 2000 which is a joint project introduced by various well-known financial institutes and companies such as Credit Suisse, and Diners Club Schweiz AG [EP06]. As a safety critical system, EP2 plays an important role, especially in the area of electronic payment management. The complexity of the EP2 system as well as the increasing software quality requirement demand a reliable, efficient, and automated testing technology which is able to assure the high quality of the product and to deal with some complicated problems such as complete and correct integration of software and hardware. In this project, we develop a prototype for testing the EP2 standard by adopting an effective and low cost testing technique, the Hardware-In-a-Loop (HIL) testing.

Used as early as the 1950s [NBAR04], HIL testing has been a useful choice in many different areas. HIL is an effective and low cost testing technology, because it provides a simulated and controlled environment for the system under test (SUT), such that the SUT only needs to communicate with the emulated components [WBM97]. Besides, test scenarios can be easily recreated.

In a typical EP2 system, there are seven major components: EP2 Terminal, Cardholder (i.e., customers), Point of Service (i.e., cash register), Attendant, POS Management System, Acquirer, Service Center and Card (See Figure 1

for more details). These components communicate with each other via specific interfaces defined by the EP2 standard. In our project, we mainly focus on the testing of the Service Center, Point of Service, and POS Management System which are significant components in the EP2 system and some related interfaces such as BE, EI, MI, and SI. We created a robust, extensible and reusable testing framework [BCL05]. It consists of the Test Case Generator, the Test Manager and the Test Reporter. The Test Case Generator is used to generate test scripts which include a list of test cases. The Test Manager executes these test cases. Finally, the Test Reporter creates a well-formatted test report according to the test result obtained from the Test Manager [BCL05]. Besides, we also make use of CEPTEST which is a remarkable EP2 development tool developed by CELSIAG, one of the companies involved in the EP2 specification.

In this dissertation, we first introduce some necessary background information related to our project. We review the EP2 standard, XML technology, Java programming language, testing technique, some available tools and reusable work. Please refer to Chapter 2 for more details. In Chapter 3, we present our implementation. In particular we focus on the XML Layer, the EP2 Component Layer, and the Testing Layer. We discuss our implementation, with some typical examples to make the content more understandable. Finally, we evaluate our implementation, summarize our project, and indicate future work in the last three chapters.

2. Background

In this chapter, we review the EP2 standard. We especially pay attention to the Service Center, Point Of Service (POS), POS Management System (PMS) and their related interfaces such as BE, EI, MI, SI. Then we will introduce XML which is the data format standard for the EP2 messages and our test scripts. We will also explain some important features of Java which make Java especially suitable for our implementation. Since testing is a crucial part of our project, we will focus on some important testing technology particularly the hardware-in-a-loop testing technique. Finally, we will present some tools and reusable work that enable us to implement our project more easily.

2.1 EP2 Standard

In an effort to promote the chip-based EMV (Europay/MasterCard/Visa integrated circuit standard) payment method particularly in the Swiss market, a joint project, eft/pos 2000 (EP2), is developed by several notable institutes and companies including Aduno SA, Diners Club International, JCB International Co. Ltd., Post Finance C Swiss Post, Swisscard C AECS AG, Telekurs Multipay, and Verband Elektronischer Zahlungsverkehr VEZ [EP06]. The EP2 standard is an open system which is based on a number of international standards and recommendations (e.g. EMV standard). So it is able to interoperate among different eft/pos infrastructures all around the world. We will give a brief introduction to the EP2 standard and then focus on the Point Of Service (POS), Service Center, POS Management System (PMS) and their related interfaces in the following subsections.

2.1.1 Electronic Payment System

Figure 1 illustrates all the roles, components and interfaces which are within the EP2 system.

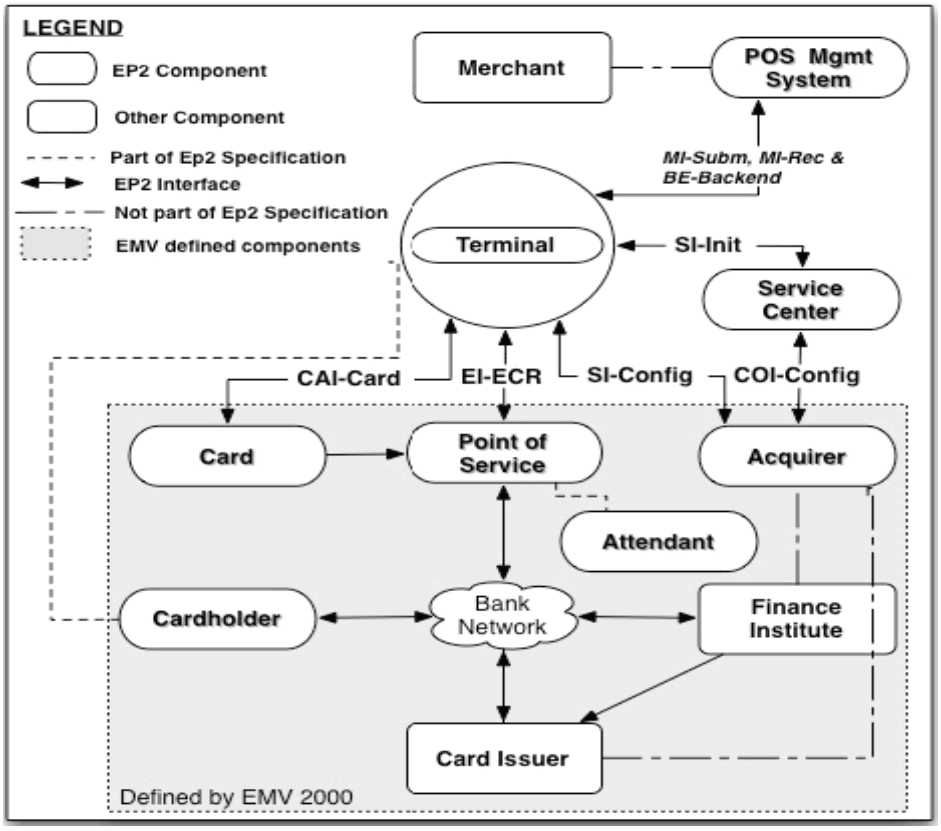

Figure 1. EP2 Payment System Model [BCL05]

There are six major components which participate in the EP2 environment [ECSC02a]:

1. Terminal:

   This processes transactions.

2. Point Of Service:

   This is a system used by customers to purchase goods or services. There are three system types: electronic cash register, attendant amount entry module and vending machine.

3. Acquirer:

   This processes electronic payment transactions.

4. Card:

   This is used by a specific cardholder to make financial transactions.

5. POS Management System:

   This is used by the merchant to manage his or her terminal population and retrieve the EFT-payment information.

6. Service Center:

   This offers configuration and maintenance services to the terminals.

As the communication center of the EP2 system, Terminal interacts with other components via specific interfaces defined by the EP2 standard. For instance, the Terminal can send and receive particular messages to and from the Service Center via the SIConfig interface. Other components are able to communicate with some certain components through corresponding

interfaces. The interfaces BE, EI, MI, and SI (Config) involved in our project are described as follows [ECSC02b]:

1. BE:

   The transaction log data which will be saved in the PMS log on the POS Management System are transmitted through the BE interface between the terminal and the POS Management System.

2. EI:

   The POS uses the EI interface to send information to the terminal for further processing. The information can be that of transactions, users, terminal activations and deactivations.

3. MI:

   The MI interface is used by the Terminal or PMS to submit transaction data to the acquirer and the acquirer will process these transaction data files and generate a reconciliation advice file (RAF) for the merchant per reconciliation period.

4. SI (Config):

   The Terminal acts as communication master to start a communication session with the responsible service center after the terminal pre-configuration for the terminal configuration via the SI (Config) interface. As communication slave, the Service Center can request the terminal configuration and initialisation data.

2.1.2 EMV and EP2

As the electronic payment application became more and more popular all around the world, Europay International, MasterCard International, and Visa International formed an organization called EMVCo, LLC in February 1999. The main role of ECVCo is “to manage, maintain and enhance the EMV Integrated Circuit Card Specifications to ensure interoperability and acceptance of payment system integrated circuit cards on a worldwide basis” [EMVCo].

In fact, the EP2 standard is an extension to the EMV standard. Based on EMV, the EP2 introduced more interfaces from various components to the terminal. Besides, the EMV standard has been an important reference for the EP2 standard. One good example is that EP2 adopts the same cryptographic approaches specified by the EMV specifications [BCL05]. In many cases, EP2 developers need to review both of these specifications to implement their applications.

2.1.3 Terminal

As the communication center of the EP2 system, the terminal specifies an interface for the cardholder or the merchant to perform financial, chip or magnetic card based transactions [ECSC02c].

An EP2 terminal is a piece of hardware which consists of a basic module (e.g. CPU, clock, memory, and communication facility), display, key pad, PIN pad, card reader, buzzer and printer. In order to ensure the terminal’s

capabilities of multi-application, software versioning, remote software download, and embedding non-EP2 applications, appropriate software architecture and an operating system would be selected for the terminal.

Some variants of the Terminal exist to process different tasks. Variant 1 is a stand-alone terminal to submit data submission files to the acquirer via the MI interface. Variant 2 is the terminal connected to an external PMS. There are two types of Variant 2: Variant 2a which is in charge of data submission processed by the terminal, and Variant 2b which is in charge of data submission processed by the PMS. For more information about the terminal please review the "Terminal Specification: General Requirements", the "Terminal Specification: Functional Requirements", the "Terminal Specification: Supplementary Requirements", and "Interface Specification" which are part of the EP2 specifications.

2.1.4 POS and EI Interface

POS stands for Point Of Service. It provides an interface for the cardholder or the merchant to execute financial, chip or magnetic card based transactions [ECSC02d].

The POS hardware contains a key pad (including a touch screen and different types of keys), a display, and a printer. The POS should conform to all related Swiss and international standards (electromagnetic compatibility, telecom compatibility, etc.) and satisfy all hardware-related requirements defined in EMV. And the POS software architecture must fulfil all the requirements of EMV 2000 and EP2.

There are three POS variants: Attendant Entry Module (AEM) which is used by the attendant to activate functions on the terminal, Electronic Cash Register (ECR) which is a system used to track the amount of money received, and the vending machine which is used to sell goods and services automatically.

The POS is connected to the terminal via a bidirectional electronic communication interface, EIECR. For different purposes, the POS is allowed to send different sorts of messages to the terminal. And the terminal will send corresponding messages back to the POS. Table 1 shows different messages allowed in EIECR and their related functions.

Function	POS → Terminal	Terminal → POS
Activate Terminal	<<Terminal Activation Notification>>	<<Terminal Activation Acknowledge>>
Deactivate Terminal	<<Terminal Deactivation Notification>>	<<Terminal Deactivation Acknowledge>>
Get Receipt	<<Receipt Request>> <<Commit Notification>>	<<Receipt Response>> <<Commit Acknowledge>>
Get Terminal Status	<<Status Request>>	<<Status Response>>
Initiate Terminal Setup	<<Init Terminal Setup Request>>	<<Init Terminal Setup Response>>

Trigger Transmission / Submission	<<Submission Request>>	<<Submission Response>>
Process Transaction	<<Init Transaction Request>> <<Transaction Request>> <<Commit Notification>>	<<Init Transaction Response>> <<Transaction Response>> <<Commit Acknowledge>>

Table 1. Messages allowed in EIECR and their functions [ECSC02b]

The EP2 Interface Specification provides detailed information for all these messages including message properties, data element of each message and sample messages. Take << Commit Notification >> message as an example. Table 2 and Table 3 indicate its properties and data element respectively. A sample message from the EP2 Interface Specification is shown in Figure 2. We will explain it in the XML section (See Section 2.2).

XML – Tag	ep2:poscomntf
Direction	POS → TRM

Table 2. Properties of the << Commit Notification >> Message [ECSC02b]

Name	XML – Tag	Condition
<POS Identifier>	ep2:POSID	
<POS Transaction Result>	ep2:POSTrxRes	
<POS Transaction Sequence	ep2:POSTrxSeqCnt	Present if the commit

Counter>		is send for processing a transaction
<Terminal Identification>	ep2:TrmID	

Table 3. Data Elements of the << Commit Notification >> Message [ECSC02b]

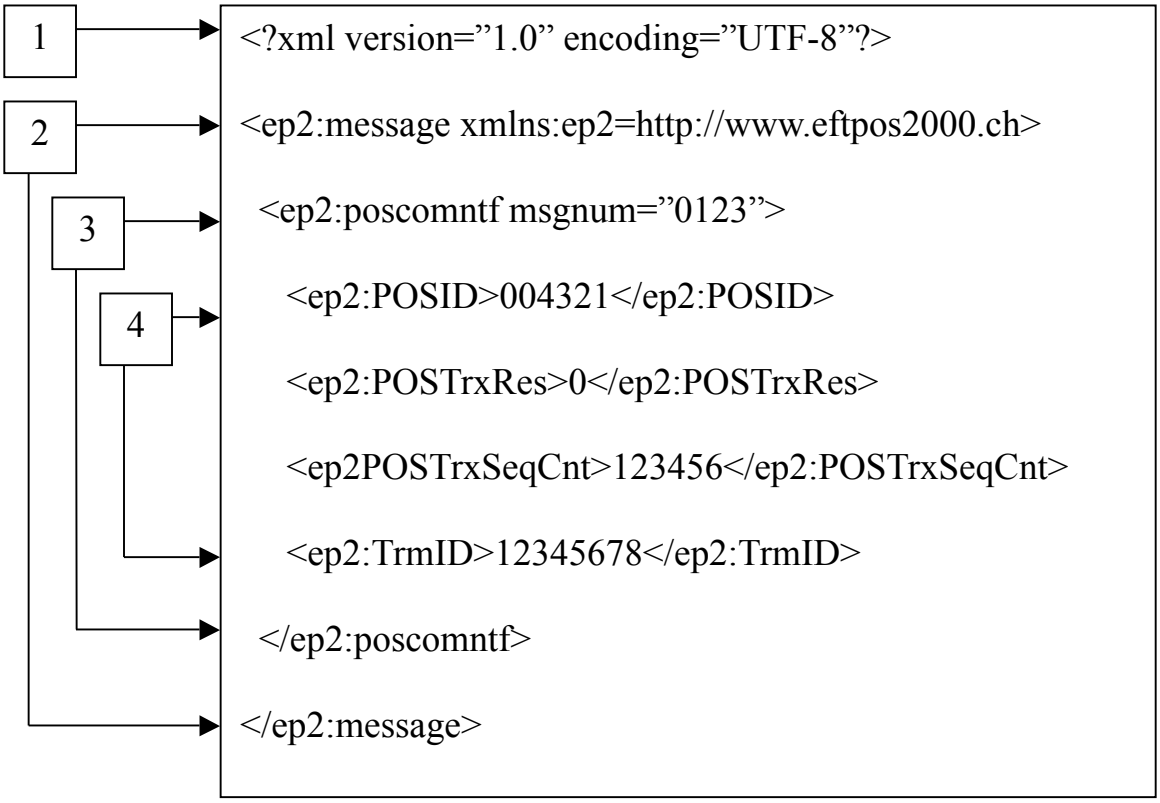

Figure 2. Sample Message of << Commit Notification >> [ECSC02b]

More information about the POS and the EIECR interface can be found in the “POS Specification” and the “Interface Specification”.

2.1.5 Service Center and SI (Config) Interface

The Service Center's main job is to configure and maintain the terminal, exchange terminal configuration data with the acquirer [ECSC02e].

The Service Center consists of two logical servers which are the configuration server and the COI server [ECSC02e]. The configuration server is used to request or download data from the terminal via the SIConfig interface. Being accessible periodically, it maintains data consistency together with the COI server. And the COI server is used to exchange data with the acquirer via the COI interface. Only the SIConfig is related to our project, so we will only discuss this interface in this subsection.

The SIConfig serves as a bidirectional interface between the service center and the terminal. Acting as communication master, the terminal is allowed to trigger a communication session with the server center. And as a communication slave, the service is able to request terminal configuration and initialisation data. Various messages are permitted by the SIConfig interface. Table 4 indicates these messages and their functions.

Functions	Terminal → Service Center	Service Center → Terminal
Configuration Data Download	<<Session Start>> <<Config Data Acknowledge>> <<Activate Config Data Notification>>	<<Config Data Notification>> <<Activate Config Data Notification>> <<Session End>>

Reset Terminal	<<Session Start>> <<Reset Terminal Acknowledge>>	<<Reset Terminal Notification>> <<Session End>>
Removing Configuration Data	<<Session Start>> <<Remove Config Data Acknowledge>>	<<Remove Config Data Notification>> <<Session End>>
Request Configuration Data	<<Session Start>> <<Config Data Response>>	<<Config Data Request>> <<Session End>>

Table 4. Messages allowed in SIConfig and their functions [ECSC02b]

Please reference the "Service Center Specification" and the "Interface Specification" for more information.

2.1.6 PMS and the BE Interface

PMS is the abbreviation for POS Management System. Merchants are able to administrate their terminal population and obtain information about the status of the eft-payments [ECSC02f].

The PMS contains three servers: a transmission server, a submission server, and a reconciliation server [ECSC02f]. The transmission server is used to retrieve and store all transaction data sent by the terminal over the BE interface. The submission server generates data submission files according to data stored on the transmission server and then sends these files to the acquirers for reconciliation via the MISubm interface. And the reconciliation

server is responsible for retrieving the reconciliation advice files created by the acquirer via the MIRec interface and recreating data submission files when exceptions happen. It also provides statistical data for controlling purposes to the bookkeeping system [ECSC02f]. We will discuss these three interfaces in this subsection.

As a bidirectional interface, the BE is chiefly used for transferring the transaction log data which will then be saved in the PMS log on the PMS. Messages allowed in the BE interface and their functions are shown in Table 5.

The MI interface is an electronic link between Terminal/PMS and Acquirer. The message <<Data Submission File Notification>> can be submitted from the Terminal or PMS to the Acquirer through the MISubm interface. There is another interface called MIRec available to the PMS. The PMS can send the message <<Reconciliation Advice File Request>> via this interface. On the other side, the Acquirer will respond with the message <<Reconciliation Advice File Response>> and <<Data Submission File Acknowledge>> respectively.

Function	Terminal → PMS	PMS → Terminal
Activate Terminal	<<POS Activation Notification>>	<<POS Activation Acknowledge>>
Deactivate Terminal	<<POS Deactivation Notification>>	<<POS Deactivation Acknowledge>>
Data	<<Data Transmission File	<<Data Transmission File

Transmission	Notification>>	Acknowledge>>
Transmit Terminal Status	<<Terminal Status Notification>>	<<Terminal Status Acknowledge>>

Table 5. Messages allowed in BE and their functions [ECSC02b]

The “PMS Specification” and the “Interface Specification” are necessary reference for implementation of PMS and its related interface introduced in this subsection.

2.2 XML

Extensible Markup Language (XML) is a simple but flexible markup language which can contain and manage any kind of structured information in a required form in many areas, especially Web applications. “All ep2 defined interfaces use XML (eXtended Markup Language) as data format standard.” [ECSC02b] Hence it is necessary to grasp basic knowledge of XML. In this section, we will give a general introduction to XML and discuss XML structure and terminology by explaining a real EP2 message defined in the EP2 specifications.

2.2.1 Overview

XML is an information processing protocol and data storage toolkit introduced by the World Wide Web Consortium (W3C) to face the challenges of large-scale electronic publishing. As a platform independent

standard, XML is widely used to make the data exchange across the network more easily and efficiently.

Being a meta-language [Eck01] and open standard [Ray01], XML is able to reserve and manage any kind of information in a format that meets your requirements. XML ensures the high quality of a document by introducing syntax, internal link checking, comparison to document models and data typing [Ray01]. Also, it supports a wide range of writing systems and symbols.

An XML document is easily understood and analyzed by humans and software because of its simple syntax and unambiguous structure. XML allows users to define their own Document Type Definition (DTD) or an XML Schema which is used for describing sets of tags and attributes to store and organize data in a desired format. Users are able to create their own tags and define the meaning for each tag (e.g. <ep2 >).

2.2.2 XML Basics

In XML, a tree-like structure and a self-describing syntax are used to manage information. A typical XML message is shown in Figure 2. There are four major constituents in a typical XML message. We will explain them in details (Please notice that our explanation corresponds the numbering in Figure 2).

The Document Prolog/Header This declares the XML version being used and the character encoding used in the document. Thus we can learn from

the prolog of our example that the message uses XML version 1.0 and UTF-8 as character encoding.

Root Element The first element immediately after the header is called the root element or document root (in our example: ep2:message). Being the first opening tag and the last closing tag within the document, the root element is the root of all other elements in the document. The XML parser can reference the root element to distinguish the beginning and the ending of a document.

The XML uses namespace to avoid element name conflict. “A namespace is a group of element and attribute names.” [Ray01] In our example, we assigned a namespace within the tag of the root element (xmlns:ep2=”http://www.eftpos2000.ch”).

First Child Elements placed in the root element are called children [Ray01]. Serving as a reference point when parsing XML, the first child element just like other element children, can have any number of children. And it can have zero or multiple attributes as well (e.g. msgnum=”0123”).

Elements XML elements are the building blocks of XML documents. The extendibility of XML elements enables XML documents to carry more information. An element can enclose its values within itself, e.g.: <ep2:POSID >004321</ep2: POSID >.

XML elements must follow this naming rule: “An element name must start with a letter or an underscore, and can contain any number of letters, numbers, hyphens, periods, and underscores” [Ray01].

2.3 Java

The programming language we used to implement this project is Java which was officially announced by Sun Microsystems in 1995. Java has become one of the most popular and powerful programming languages in the computing world because it has many important features such as platform independence, enforced exception handling, object oriented, a wide range of APIs and Libraries, generics, and so on. We chose Java as our tool for this project for the following reasons.

2.3.1 Object-Oriented

Java is a high level object-oriented programming language [ABO98]. In the object-oriented paradigm, we can analyze and solve problems by representing real-world objects with software objects and with communication among these software objects [CWH00]. The EP2 standard which we have discussed before, defines several components and communication interfaces among them. So Java is especially suitable for our project. Object-oriented programming has many advantages such as polymorphism, code reuse, and information hiding [Gui95]. For instance, we have designed an “Ep2Component” class to represent general EP2 components. This class provides general functions of all the EP2 components. A specific EP2 component can derive from this class. And we

also designed an “XmlMessage” class to represent general EP2 interfaces among EP2 components. This class mainly deals with the processing of EP2 messages. A certain interface can derive from this class. We will present more details of our implementation in Chapter 3.

2.3.2 Design Patterns

The design patterns are “description of communicating objects and classes that are customized to solve a general design problem in a particular context” [Gamma et al. 94]. In the design pattern space, there are three different kinds of patterns: creational patterns (e.g. Abstract Factory, Builder, Factory Method, Prototype, and Singleton), structural patterns (e.g. Adapter, Bridge, Composite, Decorator, Facade, Flyweight, and Proxy), and behavioural patterns (Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method, and Visitor) [Gamma et al. 94]. Creational patterns are responsible for the creation of objects. Structural patterns concern the composition of classes or objects. Behavioural patterns deal with dynamic interactions among classes or objects.

A Java programmer may have known and used some design patterns such as Iterator, Proxy, and Observer. Java programmers use Iterator to access elements of collection objects safely and efficiently. And the java.lang.reflect.proxy class can be used for Proxy pattern. The standard Java and JavaBean event model is a good example of Observer pattern.

We introduce the Singleton pattern in this section (Readers interested in other patterns can refer to [Gamma et al. 94]). Some classes should have

exactly one instance. For instance, there can be many printers but only one print spooler in a system [Gamma et al. 94]. The Singleton pattern can "ensure a class only has one instance, and provide a global point of access to it" [Gamma et al. 94]. Figure 3 shows the structure of the Singleton pattern.

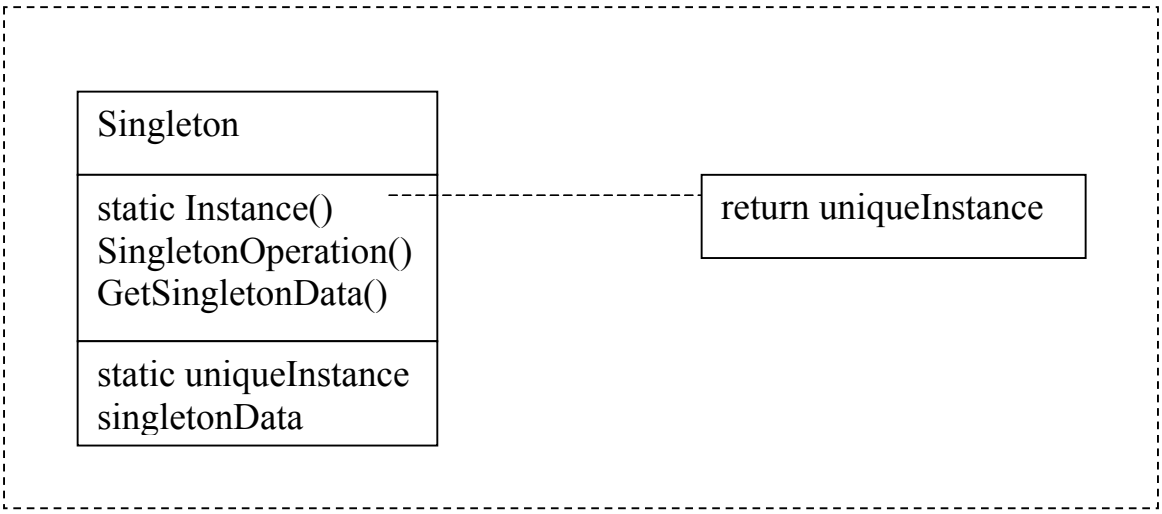

Figure 3. Structure of Singleton Pattern [Gamma et al. 94]

Gamma et al. implemented the Singleton methods in C++. We can implement them in Java:

```
Class Singleton {
    // make the constructor private
    private Singleton() {
        …
    }
    // define a final static Singleton variable
    private final static Singleton uniqueInstance = new Singleton();
    // return the uniqueInstance
```

```
    static Singleton Instance() {
        return uniqueInstance;
    }
    // other methods
    …
}
```

Java semantics make the Singleton class meet the requirements. Since the Singleton constructor is private, we can not create Singleton objects explicitly. One and only one instance object will be created before its first use. And we can only access this instance by invoking the "Instance" method.

In the object-oriented paradigm, design patterns are an extremely flexible method for software developers to implement a common solution to a common problem [BBS05]. Because of the merits provided by design patterns, computer scientists can encapsulate their experience in their works and enhance the documentation of software designs [AC98]. Mr Beng Chuan Lim did a successful project last year by adopting Java which is a programming language supporting design patterns very well. Since the close relation to Chuan's project, we can gain valuable experience from his project and reuse some of his work because of the nature and power of design patterns.

2.3.3 XML Support

In Java, there are many splendid APIs for XML processing. Because EP2 uses XML as its data format standard, Java is a useful tool for our project. In this subsection, we will briefly introduce some popular APIs for processing XML.

The Simple API for XML (SAX) Originally created as a Java-only API, SAX was the first widely used API for XML in Java and now there are several versions for some of the other programming language environments other than Java [SAX06]. SAX is an event-driven, serial-access mechanism for accessing XML documents [Arm06]. This protocol has been widely adopted by most servlets and network-oriented programs thanks to the fact that it is the fastest and least memory-intensive mechanism that is currently available for dealing with XML documents [Arm06]. Drawbacks of SAX are: 1. it is read-only API so it can not create new documents; 2. it requires more programming than other API such as DOM; 3. it is more difficult to visualize. 4. it is impossible to access or rearrange an earlier version of the document.

The Document Object Model (DOM) Unlike SAX, the Document Object Model (DOM) can not only parse XML documents but also create new ones. DOM allows users to access their documents randomly because DOM is a tree structure whose nodes contain certain components of XML documents. But this model requires more memory than SAX does.

Java DOM (JDOM) Java DOM (JDOM) is a more intuitive and powerful API for XML processing. Similar to DOM, JDOM offers the ability to read and create XML documents and uses a tree structure to model an XML document. But the usage of JDOM is much easier than that of DOM.

2.3.4 Platform Independence

Java programs are portable because all Java source code must be compiled into bytecodes which will be run on the Java Virtual Machine (JVM). Now JVM is available on almost all common operating systems such as Microsoft Windows, Mac OS, various versions of Unix and Linux systems. By using Java, we can make sure that our program and testing framework are portable and we can test our implementation on different operating systems.

2.3.5 Safe Java

EP2 is a safety critical system, so we need a safe language to implement our project. Java has been proven to be a safe language [Som04]. It enforces explicit exception handling, provides a boundary check for container objects, and adopts garbage collection technology to deal with memory allocation and reallocation. These highlights have made Java code safer and more trustworthy. Besides, Java is a high level object-oriented programming language whose source code is more readable and manageable for programmers. All of these convince us that we should use Java to implement our project.

2.4 Testing

In today's society, computers are crucial in our daily work and life. Higher and higher demand for more reliable and stable computer software has made testing a more significant stage in the software development process. We introduce some testing technology in this section.

2.4.1 Testing Fundamental

One may think that a successful testing is the one that finds no error in a software product. This view is not the truth. Testing should concentrate on finding as many as possible potential faults in a software product. According to Shari Lawrence Pfleeger who is the president of Systems/Software, Inc, there are various different classes of faults: algorithmic faults, computation and precision faults, documentation faults, stress or overload faults, capacity or boundary faults, timing or coordination faults, throughput or performance faults, recovery faults, hardware and system software faults, standards and procedure faults [Pfle01]. The objective of testing is to identify what kind of faults is causing the software failure and try to correct and remove them. It is also important to test whether the software meets its requirement specification. Most software vendors have their independent testing teams whose primary work is to test their own software products.

To be more effective in testing, we should follow certain testing principles. We can begin testing planning as soon as the requirement analysis is finished. More concrete test cases can be defined as soon as the design model is complete. In the testing process, individual units should be tested

first, and sub-systems and the entire system consisting of these units will be tested later. The Pareto principle states that 80 percent of all defects exposed in the phase of testing will probably be tracked back to 20 percent of all program units. So we should pay great attention to identifying this 20 percent of units and pay more attention to them. Because exhaustive testing is not a possibility, test cases should be designed precisely and carefully to enable them to discover as many potential errors as possible.

2.4.2 Validation and Verification

As a discipline, testing includes two processes: verification and validation [Kit95] which are defined as follows.

Validation: Confirmation by examination and provision of objective evidence that the particular requirements for a specific intended use are fulfilled [IEEE04].

Verification: Confirmation by examination and provision of objective evidence that specified requirements have been fulfilled [IEEE04].

V & V is a useful tool to examine that the software meets customers' requirements fully and correctly. Security, reliability, and maintainability of the software are also examined by V & V. Verification ensures a complete test coverage by demonstrating the correctness of a system for all its possible inputs, but it is believed to be an expensive approach because it demands a user with a good understanding of mathematical logic [KG99]. In our project, we adopt validation which is a feasible and cheaper alternative

to verification [BCL05]. Instead of testing all the allowable inputs, we test a finite number of inputs to establish the correctness of the system.

2.4.3 Testing Tactics

Software is tested from two different perspectives: white-box testing and black-box testing. White-box testing, which is also known as glass box, structural, clear box and open box testing, is a test case design method that uses the control structure of the procedural design to derive test cases [Pres05]. With little regard for the internal working of the test subject, black-box testing is chiefly concerned with the functional requirements of the system.

Without the source code for the C.A.R.U.S EP2 terminal software solution (Ccredit) and the CEPTEST EP2 simulator, we chose black-box testing as our testing tactics [BCL05]. By applying black-box techniques, we derive a set of test cases that satisfy the following criteria [MYE79]: (1) test cases that reduce, by a count that is greater than one, the number of additional test cases that must be designed to achieve reasonable testing, and (2) test cases that tell us something about the presence or absence of classes of errors, rather than an error associated only with the specific test at hand.

2.4.4 HIL

Adopted by the Defense and Aerospace industry as early as the 1950s [NBAR04], the Hardware-In-a-Loop (HIL) testing has been widely used to

verify critical systems in projects such as the design of wireless communication chipsets [SSHF04], Powertrain Control Module (PCM) [NBAR04], the power and thermal control unit of the X-ray satellite ABRIXAS [SMH99], and so on. Testers can reduce the testing cost by adopting HIL testing.

Due to the benefits providing by the HIL testing, we adopt this method as our testing approach. Including representations of the hardware (EP2 components) and test components such as test manager, a model is built up to simulate the computational environment with which the system under test (EP2 terminal) interacts. Figure 4 illustrates our HIL test model used for testing EP2 system where an EP2 terminal is the test subject. Under the instruction of the test manager, input from our input sources (EP2 components) will be sent to the terminal. The test manager will then capture and analyze the output. A loop between test subject and testing components is constructed in this process, so it is called hardware-in-the-loop testing.

It is able to extend this technique for software testing which is known as Software-In-the-Loop (SIL) testing. In our project, the SUT is the CEPTEST EP2 simulator, so it is SIL testing. It is applicable because the real EP2 components and the CEPTEST EP2 simulator communicates using TCP/IP which is a protocol that gets rid of hardware and software differences between real components and simulator [BCL05].

Offering a fast and systematic way of testing, the simulation model improves the quality and efficiency of the test. It empowers us to recreate test result without redundant exterior interferences such as unsolicited network packet

and network delay caused by network traffics. Moreover, by employing simulated components, it is much easier to reproduce specific test situation by arranging a sequence of test inputs from various components. So HIL is a good choice for our project.

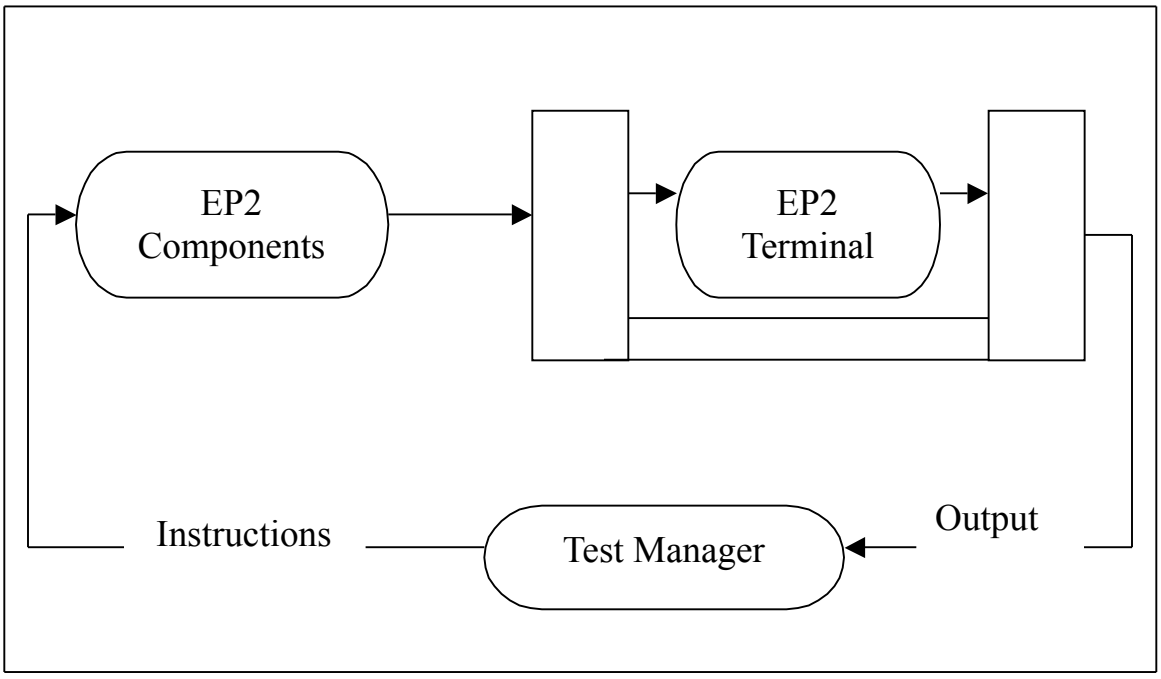

Figure 4. Hardware-In-a-Loop Setting

2.5 Chuan's Work

Mr Beng Chuan Lim produced a project which is called "Towards Hardware-In-a-Loop Testing for an International Standard of Electronic Payment System". Since there is an intimate relation between our project and Chuan's work, we can gain invaluable experience from his project and reuse some parts of his works. From top to bottom, our software is composed of five layers (Please see Figure 8 and refer to Section 3.1 for

more information): Test Applications layer, EP2 components layer, Conversation (XML) layer, Cryptography layer, and Network layer. Chuan had implemented the cryptography and network layer which we can reuse. And the Conversation, EP2 components and Test Application layers for the EP2 terminal and acquirer are also available (Please refer to Section 3.2). Chuan has indicated that interfaces for other EP2 components and further test scripts can be developed easily due to the extendable software architecture [BCL05]. For our project, we will reuse the cryptography and network layers and implement some specific EP2 components (e.g. Service Center, POS, PMS) and their related interfaces (e.g. BE, EI, MI, SIConfig) for another three layers.

Besides those works shown above, Chuan's deep insight into EP2 standard, testing technique, XML technology, Java programming, and software design principles also inspires us to some extent. For example, we grasped more information about HIL testing from his work and we learned much from his software design principles.

2.6 CEPTEST

CEPTEST is an ep2 developers test tool created by CELSI AG which is a company involved in the ep2 project. The company's expertise in EP2 technology has made CEPTEST a brilliant tool for EP2 application development.

CEPTEST is able to simulate "any component, master or slave, any communication protocol of ep2 (TCP/IP + Ethernet, ISDN, PSTN), with or

without security header and MAC, with any ep2 data elements and formats" [CEP06]. This tool has been very helpful for our project in simulation of EP2 components and investigation of the EP2communication mechanisms. We use CEPTEST to verify EP2 messages generated by our program at three different levels (XML Layer, Cryptography Layer, and Network Layer).

Figure 5 is the screenshot of CEPTEST. There are six menu selections: File, Edit, View, Parameters, Sequences, and Help. Each menu provides a kind of certain functions. For instance, we can configure the communication settings by selecting the "Communication…" in the Menu Parameters (See Figure 6). Then we can set the required setting in the "Communication Settings" window (See Figure 7). We can also set other parameters (e.g. Keys, and Message Header Parameters) in such an easy way. After proper configuration, we can use CEPTEST to test our EP2 messages at different levels.

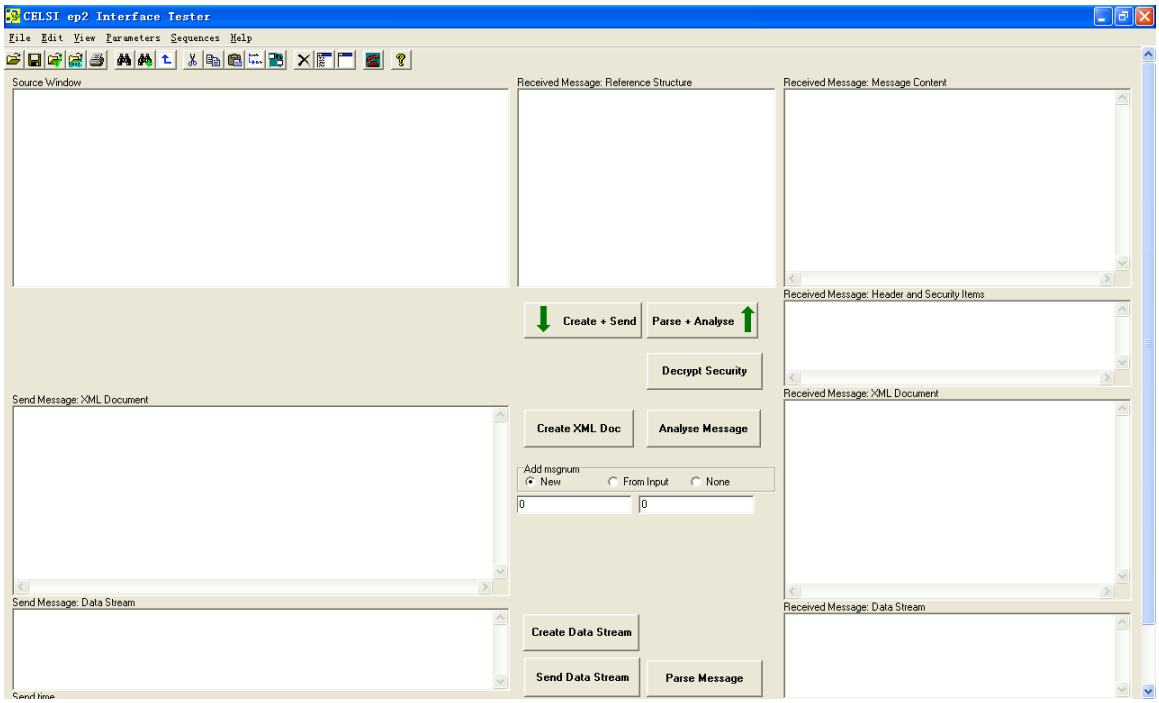

Figure 5. Screenshot of CEPTEST

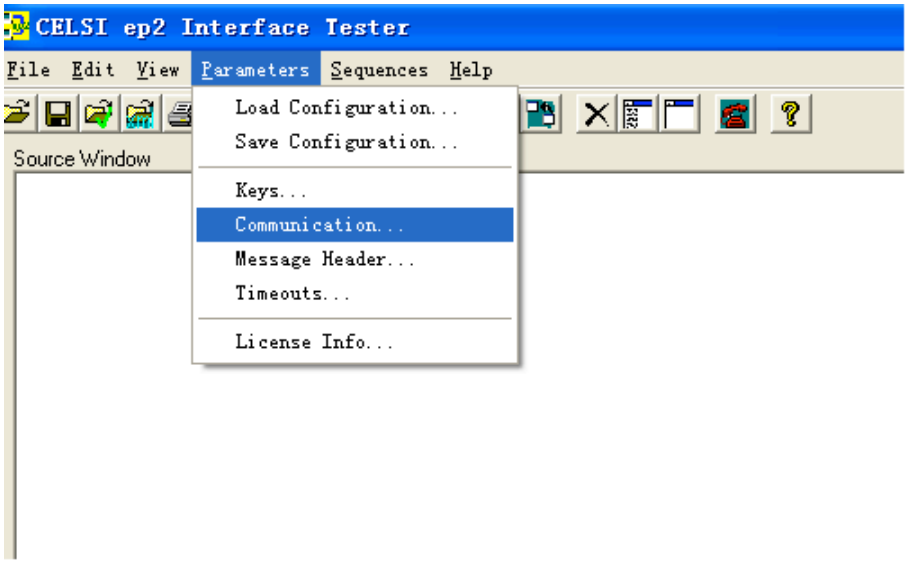

Figure 6. Menu Parameters

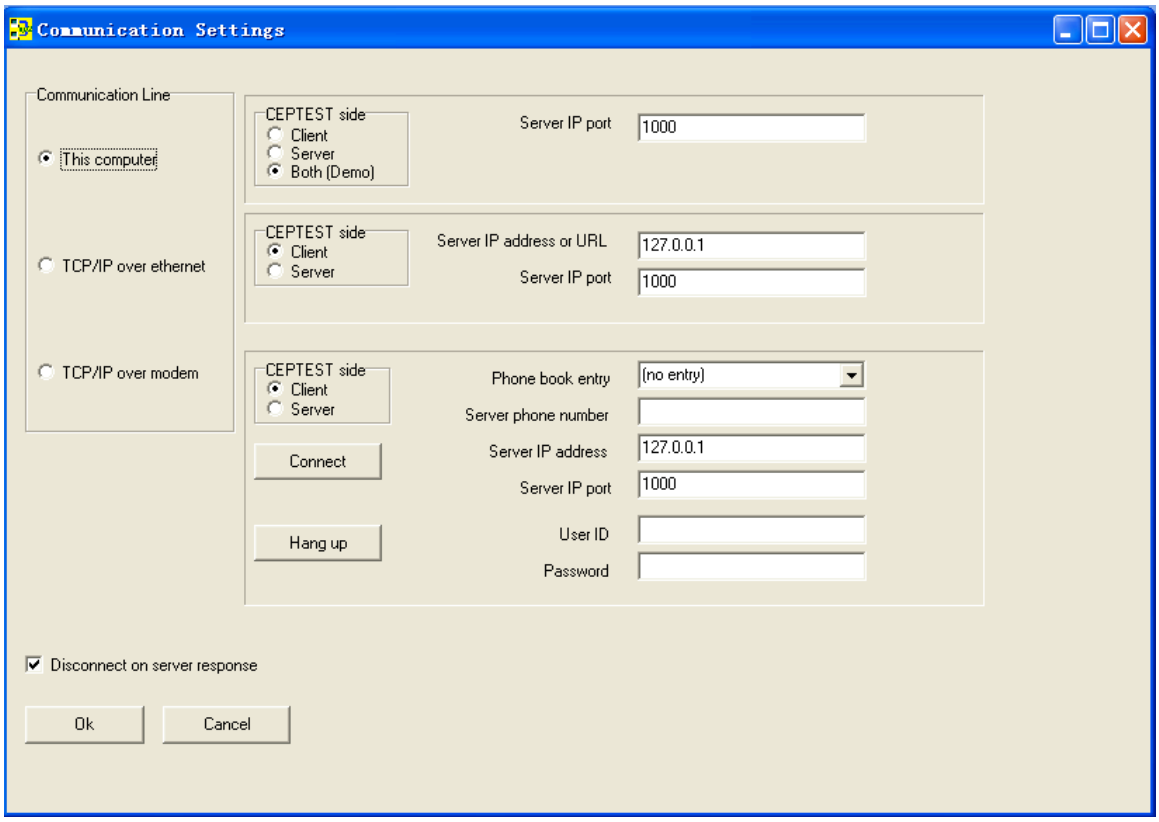

Figure 7. Communication Settings Window

3. Implementation

In this chapter, we present details of and information about our implementation. First, we introduce our software architecture and some reusable work. Then, we present the XML, EP2 Component, and Testing layers which are the core of our work. And the testing of our implementation against CEPTEST is the final part of this chapter.

3.1 Software Architecture

In this section, we describe the software architecture closely following Chuan's thesis (Please refer to [BCL05]) because it was designed by Chuan.

The EP2 system is a complex electronic payment system. It has to deal with many problems such as security, XML messages, and communication between different components. So the software architecture should be designed carefully. It must be robust, extensible, and reusable [BCL05].

Figure 8 illustrates a well-designed software architecture for our project which is a good solution to the complex EP2 system. The architecture is separated hierarchically into five layers: Test Applications Layer, EP2 Components Layer, Conversation (XML) Layer, Cryptography Layer, and Network Layer. Each of these layers solves certain tasks and provides certain specific services to the higher layers. As we can see from the following explanations, the higher layers are responsible for more abstract data because they are logically closer to users. And the lower layers are in charge of manipulation and transmission of physical data. Each layer is described as follows.

Test Applications Layer The major responsibility of this layer is testing the EP2 components. There are three components in this layer: the Test Case Generator, the Test Manager and the Test Reporter. First, the Test Case Generator will create some test scripts including a number of test cases. Then the Test Manager will execute these test cases. Finally, the Test Reporter will produce a test report in terms of the test result obtained from the test manager.

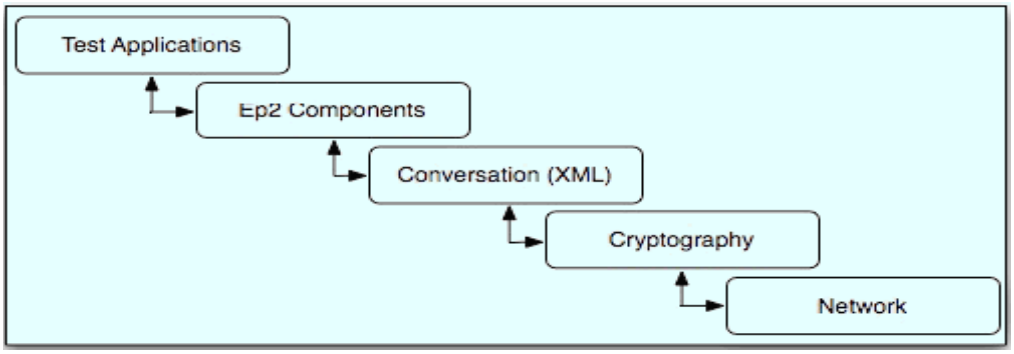

Figure 8. Software Architecture of Our Project [BCL05]

EP2 Components Layer This layer has a group of EP2 components (e.g. Service Center, POS) which provides necessary information and services to the Test Manager.

Conversation (XML Message) Layer This layer processes the EP2 XML messages. A typical conversation involves a sent message, a received message and an expected message.

Cryptography Layer This layer offers cryptographic solutions to encrypt and decrypt the data.

Network Layer The network communication protocol between EP2 components is solved by this layer.

3.2 Some reusable work

In this section, we dwell on the necessary knowledge of the network and cryptography layers which are reusable. Because these two layers are

implemented by Chuan, we use some figures from his thesis and summarize his main ideas.

3.2.1 Network Layer

An EP2 network packet has two parts: a header which contains packet-related information (e.g., packet length, specification version, and encoding type) and a payload which contains the actual EP2 message in XML format. All EP2 packets must be converted into binary format (byte) before they are dispatched to another party.

The exact EP2 header length is determined by the encoding type which indicates the security level of the communication. The EP2 standard defines three encoding types: Encoding Type 0 which is the least secure without cryptographic character, Encoding Type 1 which increases the communication security level by adopting Message Authentication Code (MAC) and a message integrity check, and Encoding Type 2 which provides the highest communication security level by using both the message integrity check and the encryption of the packet's payload before dispatching it out.

In terms of our class design, the "Config" and "ByteUtils" classes offer general services to all other classes [BCL05]. The "Config" class reserves a suit of configurations defined by users such as encoding type, and interface. The "ByteUtils" class deals with byte manipulation.

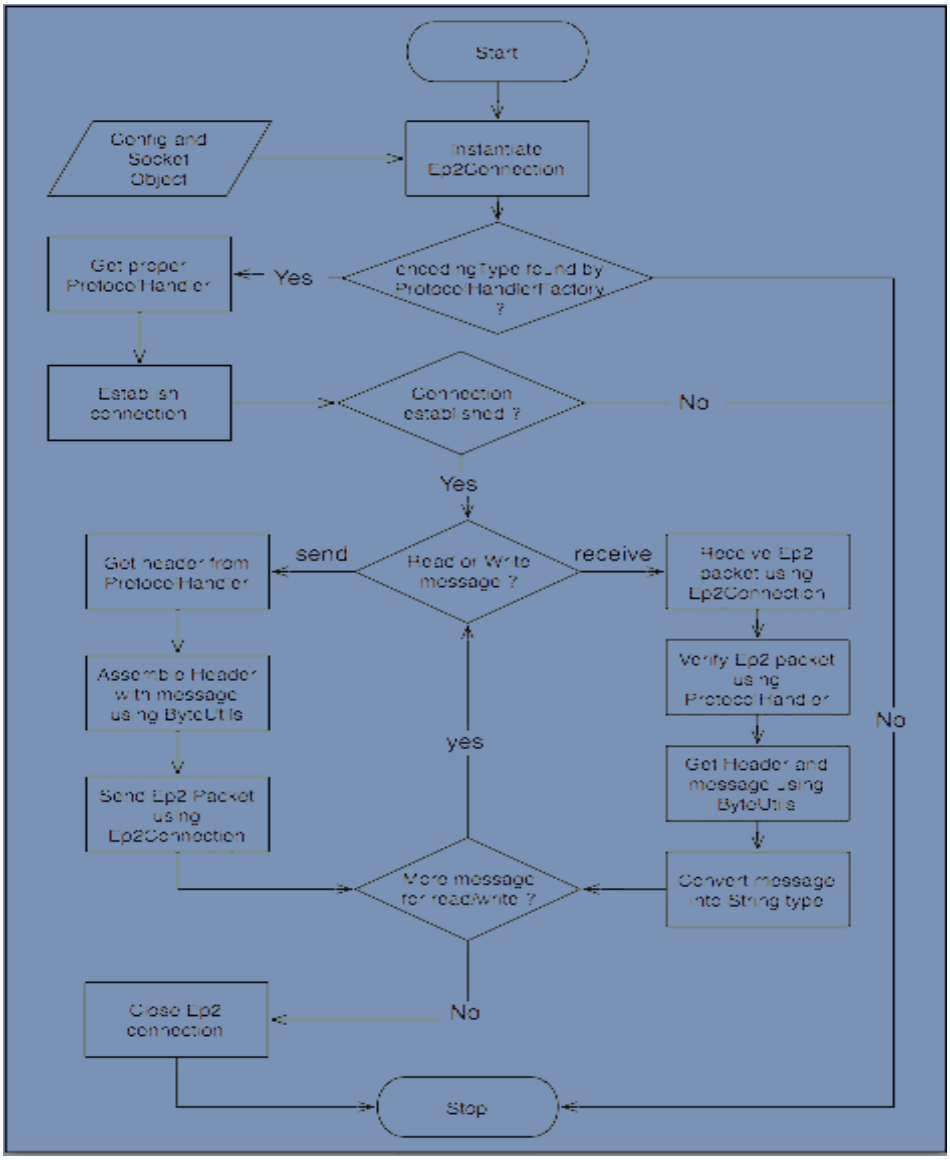

Figure 9. Netowrk Layer Workflow Diagram [BCL05]

Figure 9 is the workflow diagram of the Network Layer. The main class in the Network Layer is “Ep2Connection” which deals with the Ep2 connection

at TCP/IP level used by the Test Application to send or receive any EP2 messages. First, an “Ep2Connection” object will be initialized by the proper Config and Socket Objects. Then, the “ProtocolHandleFactory” will select and return the required “ProtocolHandler” object to format the EP2 packets according to the “encodingType”. After the connection is established, the “Ep2Connection” object can send or receive messages.

To send a message, the “Ep2Connection” will get a header from the “ProtocolHandler” object and merge the header with the actual XML message by calling the “concat” method of the “ByteUtils” class. Once the EP2 packet is merged, the “Ep2Connection” can send it to another party across the TCP/IP network.

When the “Ep2Connection” receives an EP2 packet from another party, it will use the “ProtocolHandler” to verify it. If the packet is valid, the “ByteUtils” will get the packet Header and message. Then the message in binary format will be converted into string format and be returned to the “Ep2Connection”. The following simple example shows the usage of the “Ep2Connection” class.

```
// instantiate proper Config and Socket objects
Config config = new Config();
Config.set(“encodingType”, 0);
Socket socket = new Socket(“127.0.0.1”, 6625);

// instantiate Ep2Connection
Ep2Connection connection = new Ep2Connection(socket, config);
```

```
// open the connection
connection.open();

// send a message to another party
connection.write("An EP2 message");
// receive a message
String message = connection.read();

// close the connection
connection.close();
```

3.2.2 Cryptography Layer

In the previous subsection, we pointed out that only two out of three encoding types have the cryptographic characters. They are Encoding Type 1 and Encoding Type 2. We briefly introduce some knowledge and uses for Encoding Type 1 in this section, because the implementation of Encoding Type 2 is beyond the scope of our project.

Encoding Type 1 enhances the security level of an EP2 network packet by introducing MAC into its header. In terms of ISO/IEC 9797-1, the MAC is generated by using 64-bit DES block cipher and CBC mode [EMV04]. First, we can use the "getHashValueSHA1" method to convert the EP2 message into its hash value. Then the "getPaddedData" method will be executed to do the padding process which can ensure the hash value meets the required length. Finally, the "generateMac" method will generate the MAC. And the "SessionKeyVariant" will create the Session Key Variant required by the

"generateMac". All these methods are in the "CryptoUtils" class.

3.3 XML Layer

The XML Layer is in charge of manipulating all the EP2 XML messages. In this subsection, we first use some material from Chuan's thesis to introduce the workflow and some classes of the XML Layer which are designed and implemented by Chuan. Then we describe our implementation which is based on Chuan's work.

3.3.1 Workflow Diagram

Figure 10 indicates the workflow of the XML layer. Being the core for all EP2 conversations, the "Conversation" class manages all XML messages to be sent or received. First, the "XmlMessageFactory" will help the "XmlMessage" to choose a certain subclass in terms of information provided by the "Config" object. Then, the chosen subclass will create specific "XmlMessage" object for the "Conversation". If there were no more "XmlMessage" objects, the "Conversation" will be run to send or receive the XmlMessages.

For sending messages, it will get a template from the file system and replace element values with new ones. Then the messages will be sent out. For receiving messages, the "Conversation" will build a JDOM object to extract the element names and the relevant values.

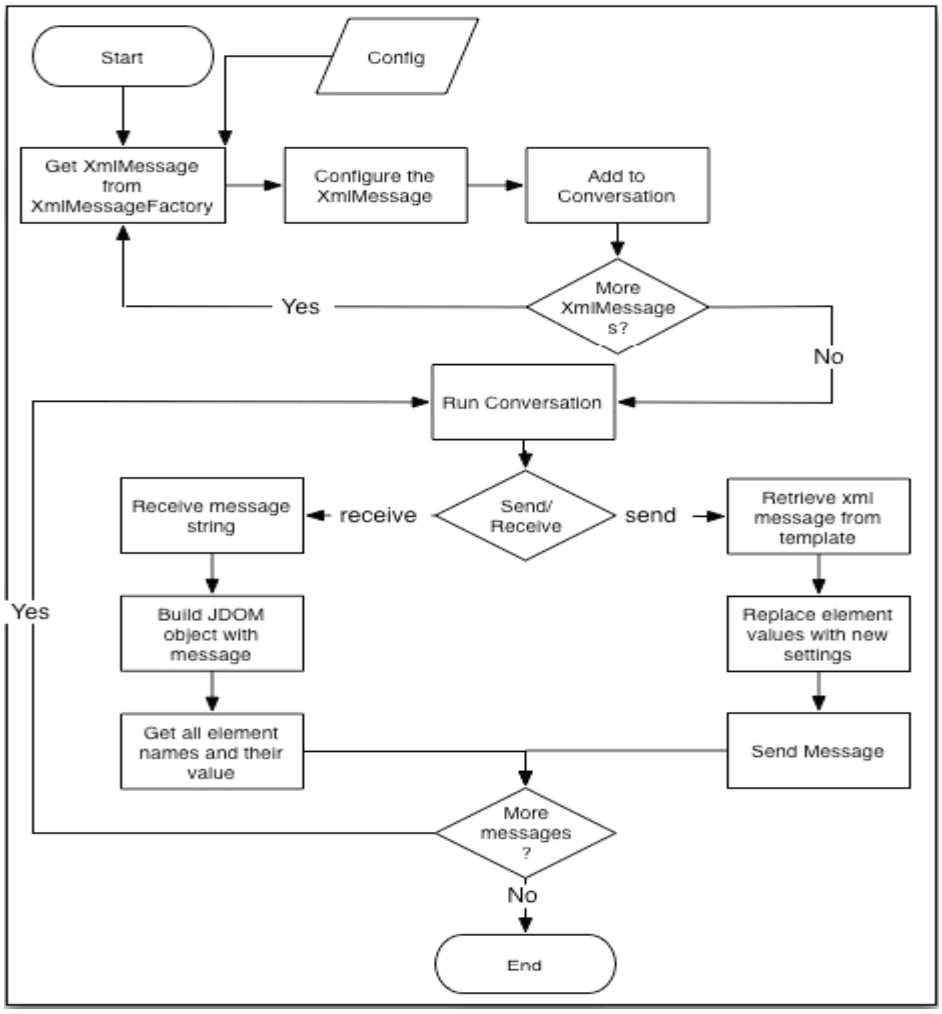

Figure 10. The XML Layer Workflow Diagram [BCL05]

3.3.2 The Classes Detail

XmlMessageFactory The mission of this class is to select the appropriate "XmlMessage" object in terms of the "Config" object. A typical example is as follows. The "Config" object provides necessary information for the

“XmlMessageFactory” object. And we can change the interface by reconfiguring the “Config” object.

```
// initialize the Config object
Config config = new Config();

// configure the Config object by providing a specific interface name
Config.set(“interfacename”, “BEBackEnd”);
// return the BEBackEnd Xml message
XmlMessageFactory.getXmlMessage(config);

// change it to a different interface EIECR
Config.set(“interfacename”, “EIECR”);
XmlMessageFactory.getXmlMessage(config);
```

XmlMessage and Its Subclasses The “XmlMessage” class can generate EP2 XML messages to be sent and process the received messages. The subclasses of “XmlMessage” take charge of XML messages of specific interfaces. Now, we have implemented the classes for six interfaces: SIConfig, SIInit, BEBackEnd, EIECR, MIRec, and MISubm.

As mentioned before, all EP2 messages are in XML format and there are many different communication interfaces which allow many different sorts of messages. So we need a feasible method to create so many different types of EP2 XML messages. Instead of using the widely used API, JDOM, which is not efficient enough to generate so many different types of XML messages, we came up with an elegant approach by introducing a model

template for each XML message in these interfaces. This method is extensible and more efficient, because we can easily add more templates to interfaces with no change to the programs, and we can readily load the templates from the file system without invoking other exterior APIs or libraries.

First, the “getPrefixedTemplate” method is invoked to obtain the storage directory of a specific message model template. To make sure the model template is of XML format, we will build it as a JDOM XML Document object before convert it into string. Then, we can replace the element values of the model template.

To receive a message, a JDOM Document object is built up by the “XmlMessage” to capture all the element names and their values, the types and numbers of the message. The “build” method is invoked to retrieve a received message. It will build a JDOM Document object. Then, it will call the “analyseType” and “analyse” methods to parse all elements from the document.

Conversation The “Conversation” class processes the dialog between EP2 components. There are four communication modes for a dialog: send, receive, send then receive and receive then send [BCL05]. Please note that, the send mode can only send a message, while the receive mode can only receive a message from other components. By setting the communication mode, a dialog is able to send a message, receive a message, or both. A “Conversation” object can reserve three “XmlMessage” objects at most: one for sending, one for receiving, and one for the expected result. One typical

usage of the “Conversation” class is demonstrated in the following example code.

```
// configure the Config object to get BEBackEnd
Config.set(“interfacename”, “BEBackEnd”);

// First Message
XmlMessage message1 = XmlMessageFactory.getXmlMessage(config);
// Second Message
XmlMessage message2 = XmlMessageFactory.getXmlMessage(config);

// set attributes
message1.set(“filename”, “dtfack”);
message2.set(“filename”, “posactack”);

// initialize conversation object
Conversation con = new Conversation();

// add messages into Conversation object
con.add(message1);
con.add(message2);

// set the communication mode
con.set(“send”);

// run the conversation
con.run();
```

3.3.3 Our Implementation

Chuan had implemented two EP2 interfaces in his project: SIInit and SIConfig. Our job is to implemented more interfaces between EP2 components. This could be easily done thanks to the nature and power of Java which is a high level object-oriented programming language. The "XmlMessage" class is the base class of all EP2 interfaces which provides all necessary methods for processing EP2 XML messages, so our interfaces could simply derive from this class. A snippet of code of the BEBackEnd interface is as follows.

```
/**
 * Override super class getPrefixed template
 */
public String getPrefixedTemplate() throws IOException, JDOMException {
    String xmlMessage = super.getPrefixedTemplate();
    return xmlMessage;
}
```

The "BEBackEnd" class is a subclass of the "XmlMessage", so it has all the functions of its super class. When this interface needs to send or receive an EP2 message, it just calls a method from its super class. The only thing we should do is just to override the "getPrefixed" method. Other interfaces could be implemented in the similar way.

3.4 EP2 Component Layer

The EP2 Component Layer has a group of EP2 component classes which are able to reserve a list of "Conversation" objects and run them. In this section, we first follow Chuan's thesis to discuss the workflow of this layer and details of these classes which were designed and implemented by him. Then we present our implementation of several EP2 components which are Service Center, POS, and PMS.

3.4.1 Workflow Diagram

The work flow is shown in Figure 11. First, the "Ep2ComponentFactory" class will construct an object of a chosen subclass of the "Ep2Component" according to the "Config" object. Then, the subclass object can add "Conversation" objects into its conversation list. If there are no more "Conversation" objects needed, we will execute all the "Conversation" in the list.

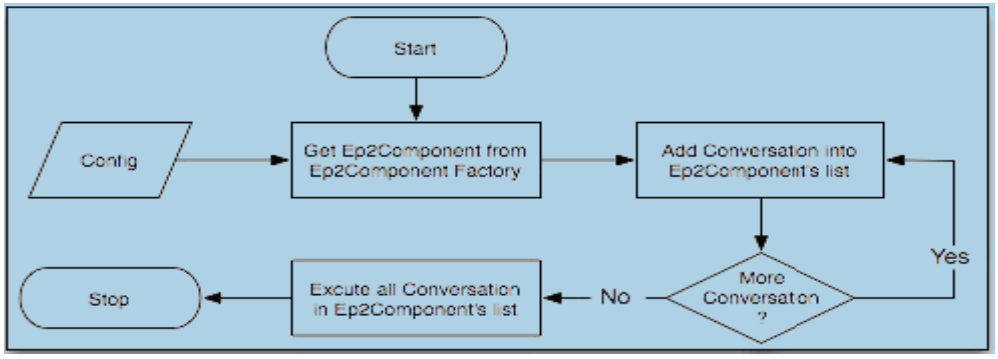

Figure 11. The EP2 Component Layer Workflow [BCL05]

3.4.2 The Classes in Detail

Ep2ComponentFactory This class is responsible for selecting the subclasses of the “Ep2Component” in terms of the “Config” object. The code below shows the usage of the class.

```
// EP2 component object
Ep2Component component;

// get the component Service Center
component = Ep2ComponentFactory.getComponent(“ServiceCenter”);

// get the component POS
component = Ep2ComponentFactory.getComponent(“POS”);
```

The “getComponent” method requires an argument which indicates the specific component. If we need the “ServiceCenter”, we can invoke the method in this way: getComponent (“ServiceCenter”). We can easily get other components in the same way.

Ep2Component and Its Subclasses These classes are designed to reserve a list of “Conversation” objects and execute them when the method “run” is invoked. The “Ep2Component” is able to run on two communication modes: Server and Client which are specified by the Ep2 standard. When it is run on the server mode, the “Ep2Component” object will establish a port for other components to connect to it. When it is run on the client mode, the “Ep2Component” will attempt to make connection to another component. Because different Ep2 components have different sets of configuration (e.g.

“ServerCenter” has “SCID” in its own configuration, “POS” has “POSID” instead), a specific subclass is designed for the corresponding component [BCL05]. Now, we have implemented subclasses for Service Center, POS, and PMS. We can understand the usage of these classes more easily by studying the following example.

```
// assume the objects will be properly initialized
Conversation con1;
Conversation con2;

// get the Service Center
Ep2Component sc =
    Ep2ComponentFactory.getComponent(“ServiceCenter”);

// configure the Service Center
sc.set(“communicationMode”, “server”);
sc.set(“serverPort”, “6625”);

// add the conversations
sc.add(con1);
sc.add(con2);

// run the conversation
sc.run();
```

We created an “Ep2Component” object and constructed it as “ServerCenter” by providing the particular argument to the “getComponent” method of the

“Ep2ComponentFactory” class. Then we customized the communication setting and add conversations to the object. Finally, we can run these conversations.

3.4.3 Our Implementation

We implemented Service Center, POS, and PMS based on Chuan’s work. All EP2 components are subclasses of the “Ep2Component” class. We explain the implementation of the “ServiceCenter” in this subsection. Other EP2 components were implemented in the similar way.

The most important method in “ServiceCenter” is the “run” method. Part of the code of the “run” method of the “ServiceCenter” is as follows.

```
public void run() throws Exception {
    …
    if(config.get(“communicationMode”).equals(“client”)) {
        logger.info(config.get(“name”) + “(Service Center) ” +
            “Running in Client Mode”);
        clientMode();
    } else if(config.get(“communicationMode”).equals(“server”)) {
        logger.info(“Service Center Running in Server Mode”);
        serverMode();
    }
    …
}
```

The “run” method will run the Service Center in client mode or server mode according to the “communicationMode” indicated by the Config object. The “clientMode” method will be invoked if the “communicationMode” is client, while the “serverMode” method will be invoked if the “communicationMode” is server. Part of the “clientMode” and “serverMode” are shown below.

```
private void clientMode() throws Exception {
    …
    Socket socket;
    int port;

    try {
        port = Integer.parseInt(config.get("port"));
    } catch(NumberFormatException e) {
        logger.error("clientMode()", e);

        logger.error("Client Invalid Port. Revert to default port:6625");
        port = 6625; // if the port is invalid, revert to default
    }
    socket = new Socket(config.get("ip"), port);
    logger.info("Connection Established at " + config.get("ip") + " port " +
    port);
    // run all the event
    runEvent(socket);
    …
```

```
}

private void serverMode() throws Exception {
    …
    ServerSocket serverSocket;
    Socket socket;
    int port;

    try {
        port = Integer.parseInt(config.get("serverPort"));
    } catch(NumberFormatException e) {
        logger.error("serverMode()", e);

        logger.error("Invalid Port. Revert to default port:6625");
        port = 6625; // if the port is invalid, revert to default
    }

    serverSocket = new ServerSocket(port);
    logger.info("Service Center is starting with server mode at port "
            + port);
    // once client accepted the connection
    socket = serverSocket.accept();

    // run all the event
    runEvent(socket);
    // close the server socket
    serverSocket.close();
```

```
    …
    }
}
```

The “clientMode” method tries to establish a connection to the specified server. Once the connection is established successfully, it runs all the events. The “serverMode” method establishes a port for other parties to connect to. When a connection is accepted, it runs all the events. All the related IP and port information are provided by the “Config” object. Part of the “runEvent” method is shown below.

```
private void runEvent(Socket socket) throws Exception {
    …
    Ep2Connection connection;
    Iterator it;
    // this number will be used through out all the messages
    int msgnum = 0;
    connection = new Ep2Connection(socket, config);

    connection.open();

    // playing all the conversation
    it = conversationList.iterator();
    logger.info(conversationList.size() + " conversation(s) for " +
    config.get("name") + "(Server Center) to run");
    while(it.hasNext()) {
        Conversation con = (Conversation) it.next();
```

```
            con.setMsgnum(msgnum);
            con.run(connection);

            if(con.getMsgnum() > msgnum) {
                msgnum = con.getMsgnum();
            }
        }

        connection.close();
        …
}
```

The “runEvent” method makes an Ep2Connection according to the “Socket” object. Once the “Ep2Connection” is open, all conversations are run.

3.5 Testing Layer

There are three major components in this layer: the Test Case Generator, the Test Manager, and the Test Report Generator. In this section, we summarize the Testing Layer workflow and discuss these three major components according to Chuan’s thesis. Then we introduce how to use this testing framework.

3.5.1 Workflow Diagram

The Test Application Layer workflow is shown in Figure 12. In order to accomplish the testing, the “TestGenerator” will first generate a test script

and send it to the “TestManager”. Then, the “TestManager” will run the test cases contained in the newly generated test script. Finally, the “TestReporter” will create a well-formatted test report according to the test result from the “TestManager”.

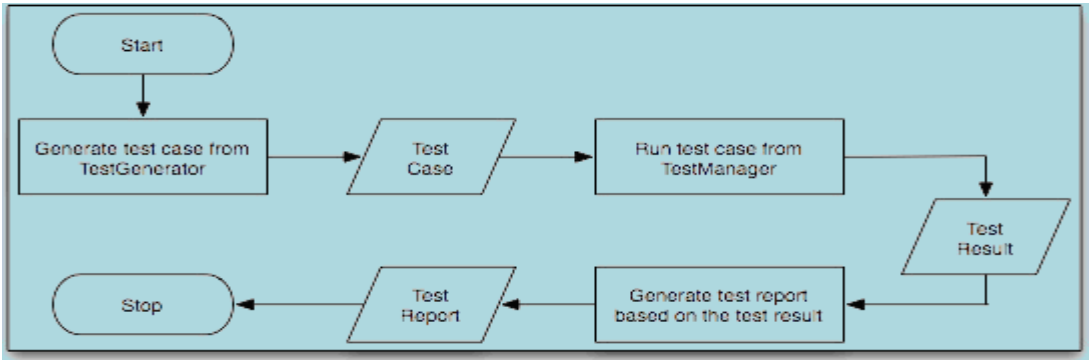

Figure 12. The Test Application Layer Workflow Diagram [BCL05]

3.5.2 Test Case Generator

The Test Case Generator’s main function is to generate test scripts. We will first introduce the test script structure. Then, we will describe the Test Case Generator workflow. Finally, we will present important classes of the Test Case Generator.

Test Script Structure In order to automate the testing procedure, we designed an adaptable test script structure which is in XML format and is used to offer instructions to the Test Manager. A typical test script contains three groups of content: basic information (e.g. test name, its description, the file path of the test result), a list of Ep2 components and their configurations,

and a list of events [BCL05]. There are two parts in every event: a source which will run all the conversations of an event, and a target which will offer necessary communication information to the source [BCL05].

Workflow Diagram Figure 13 illustrates the workflow of the Test Case Generator.

The Test Case Generator starts with the main menu which consists of six options: Test Script Menu, Test Information Menu, Ep2 Component Menu, Event Menu, and Write Test Script. Users can choose a function they want to execute. For instance, users can choose the Ep2 Component Menu to add a new component to the test component list.

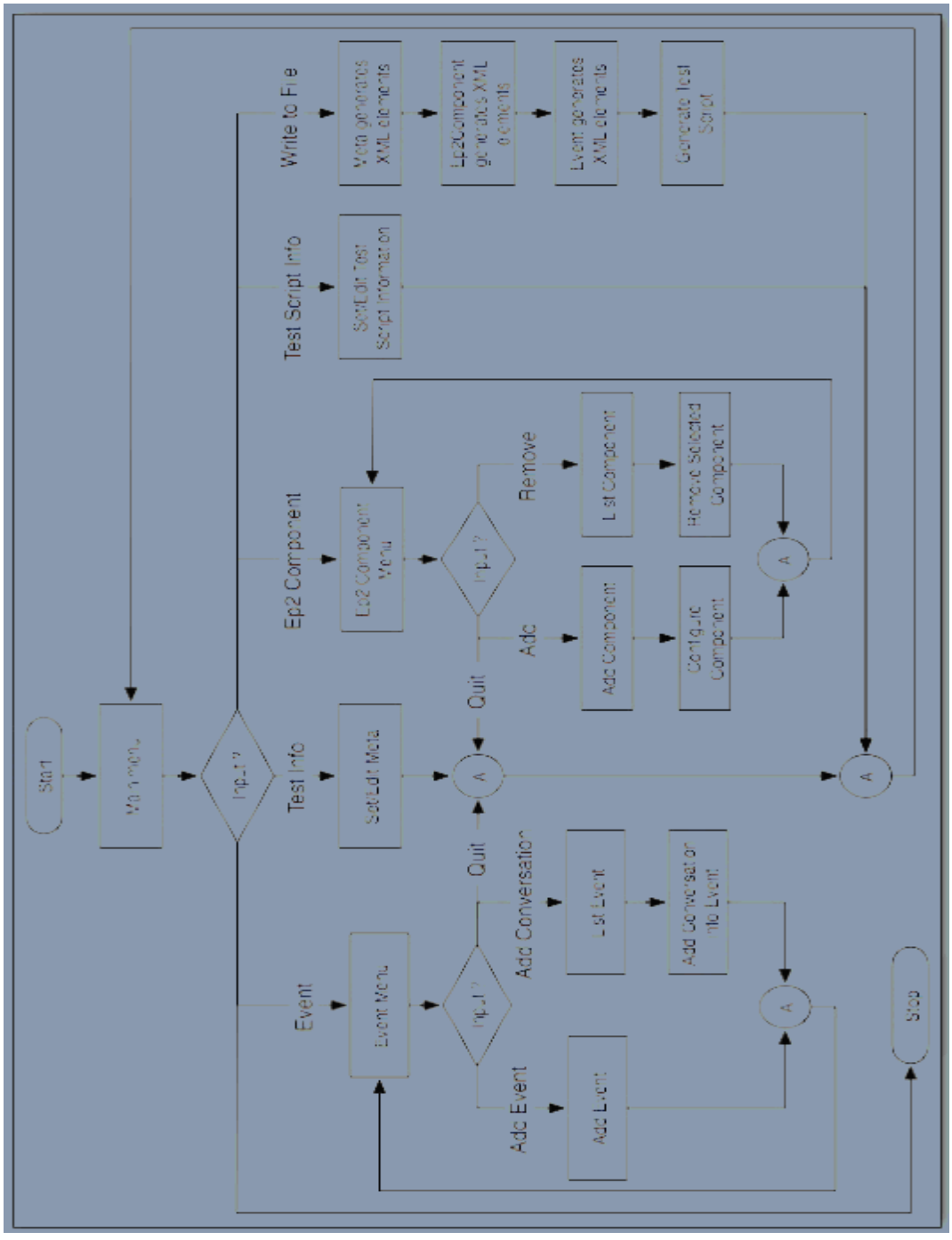

Figure 13. The Test Case Generator Workflow Diagram [BCL05]

The Classes in Details The most significant class of the Test Case Generator is the “TestCreator”.

The “TestCreator” enables us to create test scripts more easily and more efficiently. It provides a text-based interface for users to select their required functions.

3.5.3 Test Manager

The Test Manager is used to execute the test cases and collect the test results which will be used by the Test Reporter. In this section, we will introduce the test result structure, the Test Manager workflow, and some important classes.

Test Result Structure The test result structure is also in XML format and it is similar to the test case structure. A typical test result contains three sets of information too. The first two sets of information held in the test result (the basic information and the Ep2 component configurations) are the same as those in the test script. The third set of information is the test result. The test result script will be used by the Test Reporter to generate a more user-friendly test report.

Workflow Diagram Figure 14 is the diagram of the Test Manager workflow. First, it will acquire the test script’s file storage directory and select a test script. Then the selected test script will be loaded to configure some necessary settings for running the test. A test result will be generated

after the test.

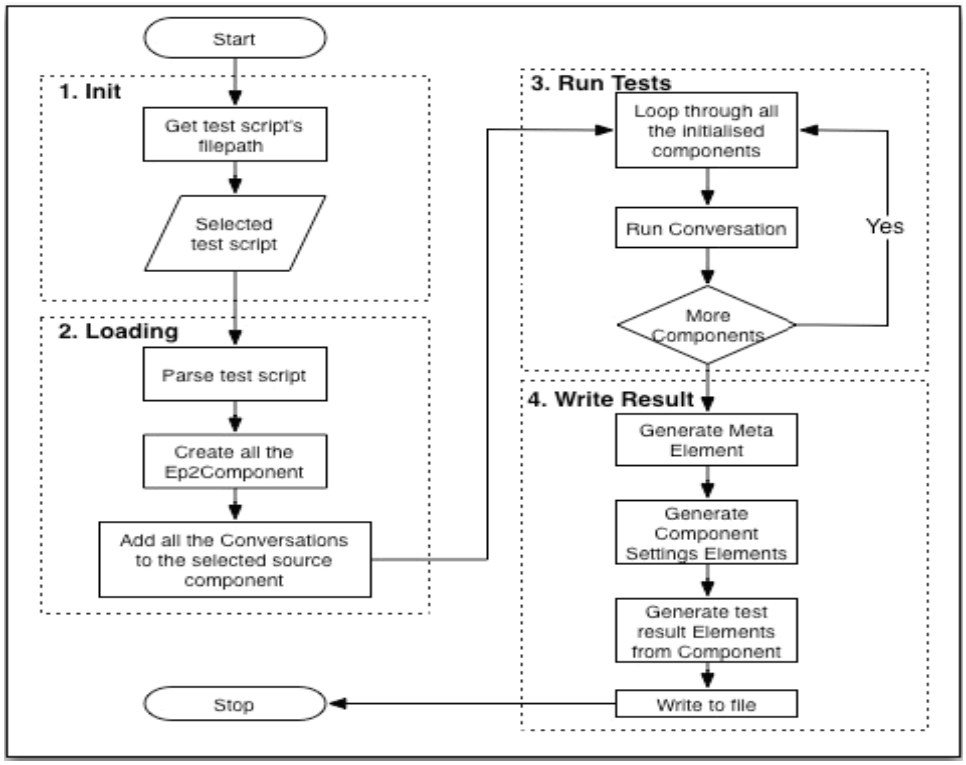

Figure 14. The Test Manager Workflow Diagram [BCL05]

The Classes in Details The "TestManagerConsole" and "TestManager" are two major classes in the Test Manager.

As the visible face of the Test Manager, the "TestManagerConsole" class's primary duty and function is to supply a text-based interface for users to choose a test script, and run it automatically. It ensures that the user specified file storage directory is valid and the selected test script is a well-

formatted XML file.

The “TestManager” class is the key class of the Test Manager, because it is in charge of the actual testing. It has three major tasks: parse and generate objects from a test script, run the test cases, and generate the test result script [BCL05].

3.5.4 Test Reporter

The Test Reporter is a flexible tool to create a user-friendly test report to the audiences. We will present the Test Reporter workflow and some significant classes in this subsection.

Workflow Diagram Before the test report is generated, the Test Reporter allows users to edit test report settings and choose an XSL template. Users can also select a required output format such as PDF or PostScript. Figure 15 shows the workflow of the Test Reporter.

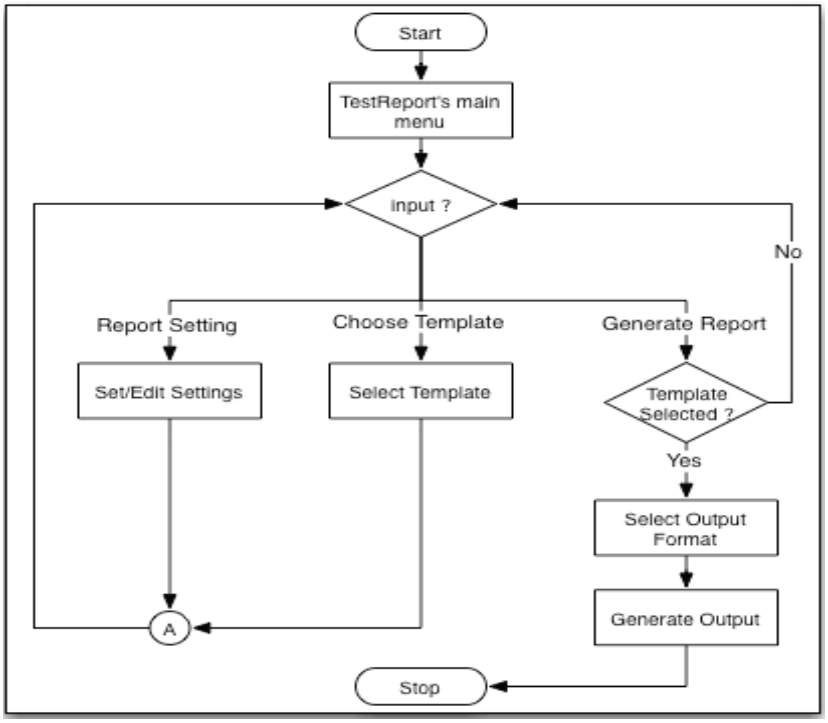

Figure 15. The Test Reporter Workflow Diagram [BCL05]

The Classes in Detail The "TestReporter" class is the most critical class in the Test Reporter.

The "TestReporter" provides a multi-choice menu which contains three options for users to select their required functions. Users can customize the report configuration such as setting the report name, specifying storage location, selecting an XSL template. Besides, users will be asked to choose an output format (e.g. awt, mif, pcl, pdf, ps, svg) before the test report is generated.

3.5.5 Using the Testing Framework

Our testing framework is an adaptable and easy-to-use testing tool. First, we use the Test Creator to generate test scripts. The Test Creator main menu offers six options for us (See Figure 16). Their functions are described as follows.

To quit: We input “-1” to exit the Test Creator.

Test Script Menu: we can set the file path used to store the test script and the file name of the test script.

Test Information Menu: we can set the basic information such as the test name and the test description.

Ep2 Component Menu: we can add or remove EP2 components (See Figure 17)

Event Menu: we can add events to the test sequence.

Write Test Script: A test script will be created when we select this option.

A typical test script is shown in Appendix A.

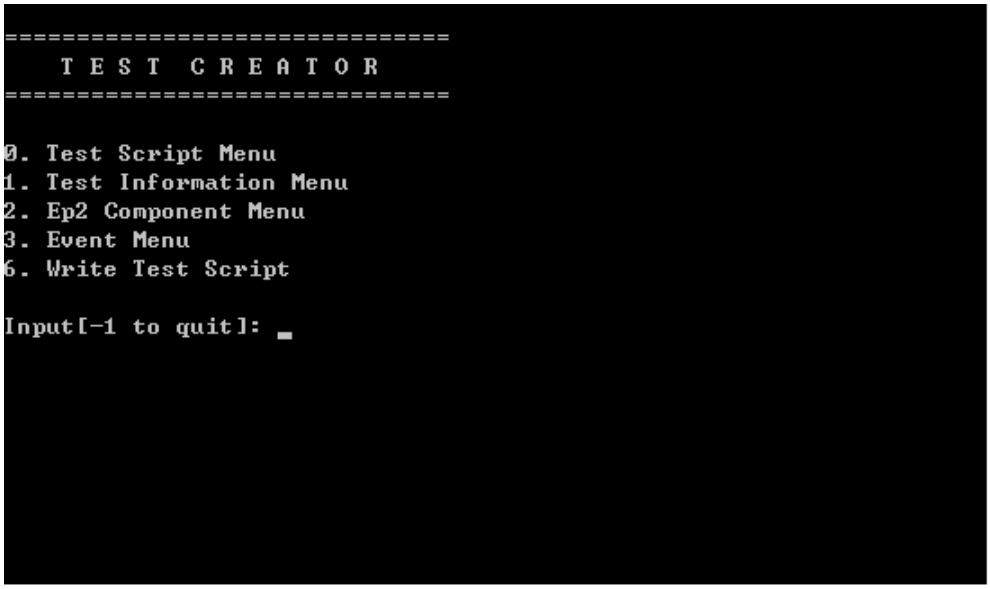

Figure 16. Screenshot of the Test Creator Main Menu

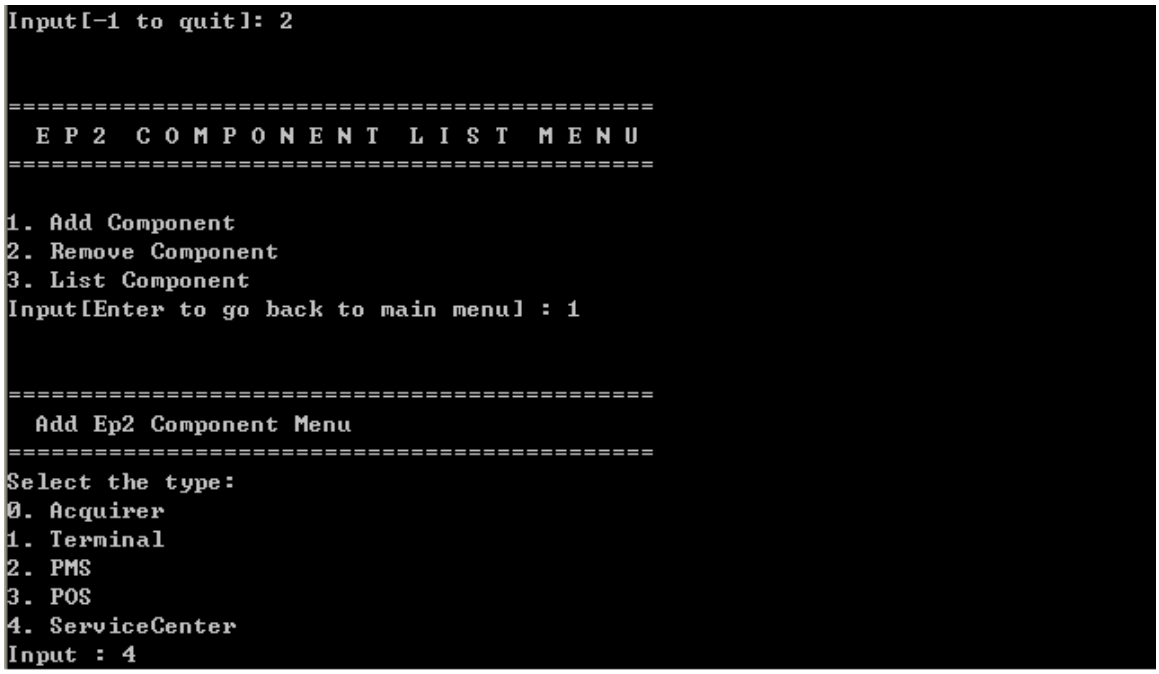

Figure 17. Screenshot of Ep2 Component Menu

Once test scripts are created, we can run the Test Manager to carry out the actual testing. We only need to provide the directory of test scripts and choose the desired test script to be run (See Figure 18).

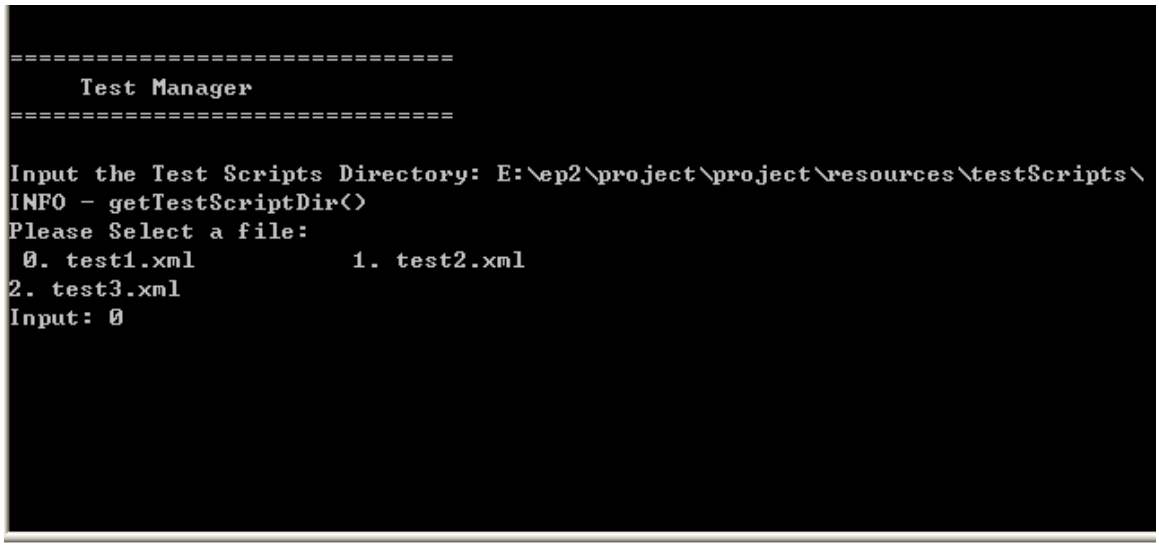

Figure 18. Screenshot of Test Manager

Finally, we can run the Test Reporter to generate test reports in the desired format. The main menu of the Test Reporter is shown in Figure 19. Once we finish the proper configuration, we can generate a test report.

Figure 19. Screenshot of the Test Report Main Menu

3.6 Testing the Implementation against CEPTEST

CEPTEST is an EP2 development tool created by CELSI AG which is a company involved in the ep2 project. It enables us to test our EP2 application more easily and efficiently. We have used CEPTEST frequently to test our EP2 implementation.

We have used CEPTEST to test three layers of our software: the XML Layer, the Cryptography Layer, and the Network Layer. These tests ensure that our program works properly. Out program can generate well-formed EP2 messages and convert them into their corresponding binary format whatever encoding type is used. And our program can parse and decrypt a received message correctly.

For instance, we can create a dtfack.xml message of the BEBackEnd interface. We can input the content of the message into CEPTEST's "Send Message: XML Document" window after the CEPTEST is properly configured. We then can convert it into its binary format. If everything is valid, we can send it out. Figure 20 shows the sent message in the CEPTEST.

After the message is received, CEPTEST is able to parse and analyse the message. Figure 21 and Figure 22 illustrates the situation of the message received.

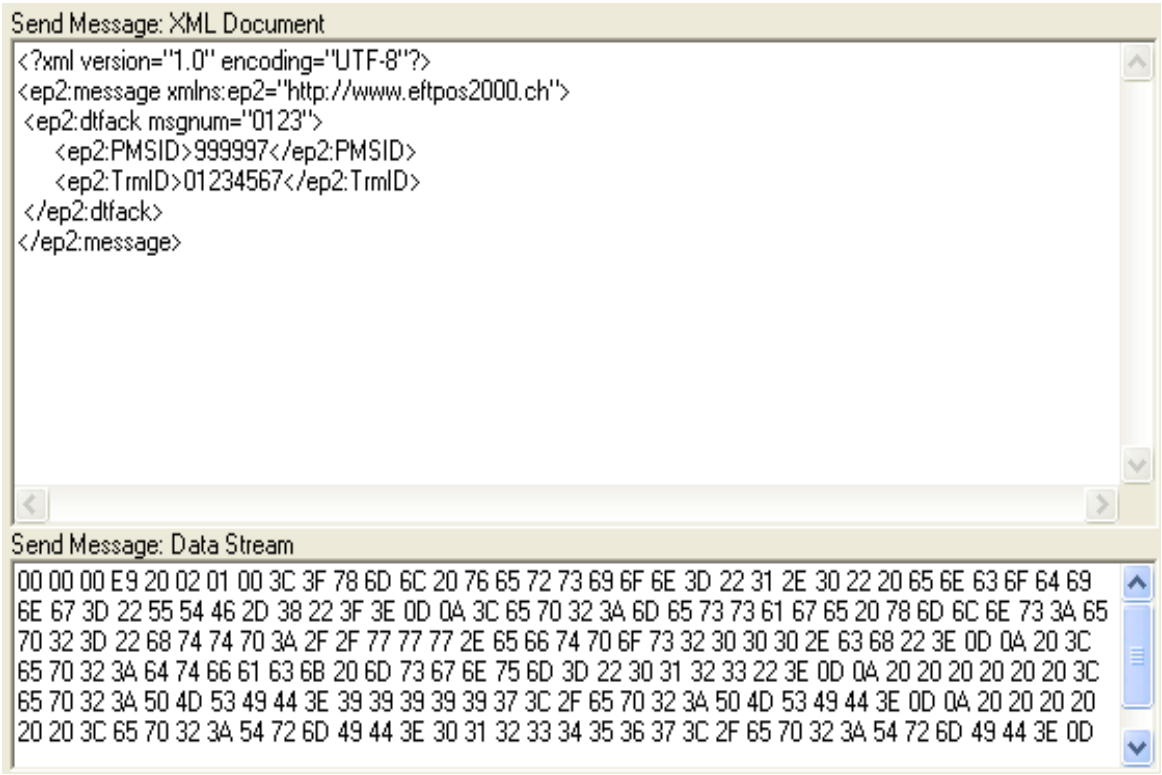

Figure 20. An example of sent message

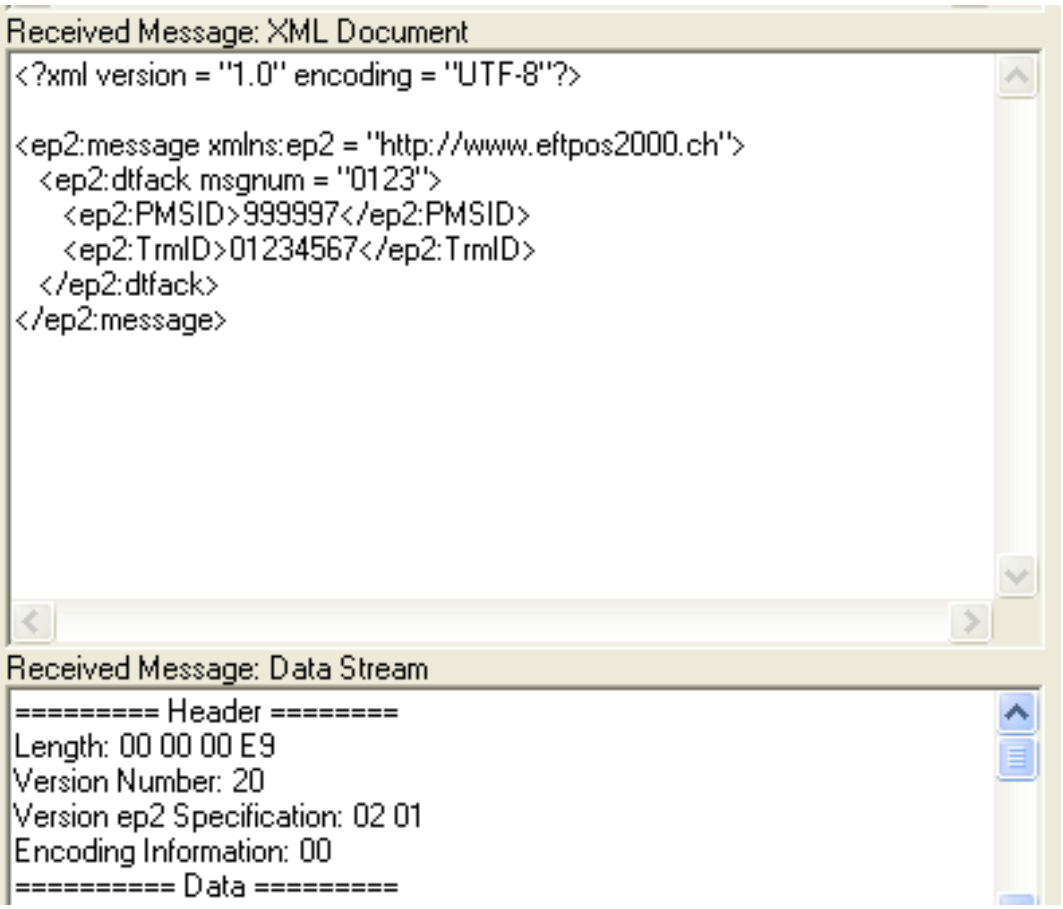

Figure 21. The received message

Figure 22. Analysis of the received message

4. Project Evaluation

Our project has three primary goals: 1. to implement some EP2 components and their related interfaces based on the previous related project. 2. to test our implementation with our Hardware-In-a-Loop testing framework. 3. to test our implementation against CEPTEST. In many ways, we have achieved these three goals completely and successfully.

We have implemented three EP2 components which are the Point Of Service (POS), the Service Center, the POS Management System (PMS) and some related interfaces such as BEBackEnd, EIECR, MIRec, MISubm, and SIConfig. We used Java as our programming language because it is an excellent development tool. Java's many important features make it a programming language especially suitable for our project. Java is a portable and object-oriented high level programming language with design pattern feature. There are also many splendid APIs for XML processing in Java. All these merits convince us to use Java to implement our project. By using Java, we can easily reuse some of Chuan's work and efficiently develop our own work.

We have designed a robust, extensible, and reusable testing framework to test our implementation. The testing technology we adopted is the Hardware-In-a-Loop (HIL) testing, which has been a widely used testing approach. It enables us to improve the quality and efficiency of our tests. Our testing framework contains three components: the Test Case Generator, the Test Manager, and the Test Reporter. We are able to test our

implementation easily with this flexible testing framework. Also, we have designed two well-formed structures for the test script and the test result script.

As a useful and handy development tool for EP2 applications, CEPTEST has been heavily used for testing our implementation. Besides simulating all EP2 components, CEPTEST is able to send, receive, analyse and display messages from any EP2 component. We used CEPTEST to verify EP2 messages generated by our program at three different layers (e.g. the XML Layer, the Cryptography Layer, and the Network Layer).

We have produced a good project and achieved its main goals, but we could do more to improve our implementation if we had more time. We can replace the text-based console menu of our testing framework with a more user-friendly graphical one. Users can generate and view the test script in a more friendly way, so the test script generation can become more efficient.

We tested our implementation mainly on Microsoft Windows XP Home operating system. We can carry out more tests on other different operating systems such as Mac OS, Free BSD, Solaris OS, and so on. Because our testing framework was implemented in Java which is portable, we can easily accomplish our testing on these different operating systems. By doing so, we can assure the high reliability and adaptability of our implementation.

5. Summary

We have developed a hardware-in-a-loop testing framework for the eft/pos 2000 system.

We studied the EP2 standard carefully. We especially paid more attention to the POS, Service Center, PMS and their related interfaces such as BE, EI, MI, and SI. We studied the functional and non-functional requirements of these components and the EP2 messages allowed for these interfaces. The thorough understanding and deep insight of the EP2 standard help us to carry out a better implementation.

We also reviewed some necessary background knowledge related to our project. With many useful features, the objected-oriented high level programming language, Java, empowers us to reuse some previous work and develop our own work more easily and quickly. XML has been proven to be an excellent technology for processing information. We used XML as the data format of our test scripts and test result scripts. This makes our framework more flexible.

Our hierarchical software architecture is an extensible and manageable solution to this project. It is a good example of the “divide and conquer” idea. There are many benefits in using this layered software architecture such as code reuse and parallel development.

The Hardware-In-a-Loop (HIL) testing is a critical testing technology

adopted by our project. Our HIL testing framework helps us to validate our EP2 implementation in a high-quality and efficient way. We also used CEPTEST to verify our implementation.

In all, we have accomplished a good project and achieved our main goals. And the most important thing is that we learn much from this project especially the testing technology and very much enjoy carrying out such an interesting project.

6. Future Work

As mentioned before, we have implemented several interfaces of the EP2 system. More interfaces such as FE and COI should be implemented in the future. This will not be a difficult job thanks to our extensible software architecture. Also, we have implemented two encoding types out of three. We need to develop the Encoding Type 2 too. Finally, we need to connect our software to the EP2 terminal hardware in the future.

7. References

[ABO98] Rebert K. Allen, Kevin Bluff, Annette B. Oppenheim. Jumping Into Java: Objected-Oriented Software Development for the Masses. Proceedings of the 3rd Australasian conference on computer science education ACSE '98. ACM Press, 1998.

[AC98] Ellen Agerbo, Aino Cornils. How to preserve the benefits of design patterns. ACM SIGPLAN Notices, Proceedings of the 13th ACM SIGPLAN conference on object-oriented programming, systems, languages, and applications OOPSLA '98. ACM Press, 1998.

[Arm06] Eric Armstrong and a world-wide cast of contributors. Working with XML: The Java/XML Tutorial. http://java.sun.com/xml/tutorial_intro.html. Visited date: Sept., 2006.

[BBS05] Alex Blewitt, Alan Bundy, Ian Stark. Automatic Verification of Design Patterns in Java. Proceedings of the 20th IEEE/ACM International conference on automated software engineering ASE '05. ACM Press, 2005.

[BCL05] Beng Chuan Lim. Towards Hardware-In-a-Loop Testing for an International Standard of Electronic Payment System, Master Thesis, Swansea University, 2005.

[CEP06] CEPTEST's homepage. http://www.celsi.ch/eftpos_ceptest.htm. Visited date: Sept. 2006.

[CWH00] Mary Campione, Kathy Walrath, Alison Huml. The Java ™ Tutorial: A Short Course on the Basics, 3rd Edition. Addison-Wesley Professional, 2000.

[Eck01] Robert Eckstein. XML Pocket Reference, Second Edition. O'Reilly, 2001.

[ECSC02a] EP2 Consortium. System Specification, eft/pos 2000 Specification Version 1.0.1, 2002

[ECSC02b] EP2 Consortium. Interface Specification, eft/pos 2000 Specification Version 1.0.1, 2002.

[ECSC02c] EP2 Consortium. Terminal Specification: General Requirements, eft/pos 2000 Specification Version 1.0.1, 2002.

[ECSC02d] EP2 Consortium. POS Specification, eft/pos 2000 Specification Version 1.0.1, 2002.

[ECSC02e] EP2 Consortium. Service Center Specification, eft/pos 2000 Specification Version 1.0.1, 2002

[ECSC02f] EP2 Consortium. PMS Specification, eft/pos 2000

Specification Version 1.0.1, 2002.

[EMV04] LLC EMVCo. EMV Integrated Circuit Card Specifications for Payment Systems (Book 2 - Security and Key Management). EMVCo, LLC, 2004.

[EMVCo] Official Website of EMVCo. http://www.emvco.com/about.asp. Visited date: Sept. 2006.

[EP06] Official Website of eft/pos 2000. http://www.eftpos2000.ch/. Visited date: Sept. 2006.

[Gamma et al. 94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley Publishing Company, 1994.

[Gui95] José de Oliveira Guimarâes. The Object Oriented Model and Its Advantages. ACM SIGPLAN OOPS Messenger. ACM Press, 1995.

[IEEE04] IEEE Standard 1012 – 1998, IEEE Standard for Software Verification and Validation. IEEE, 2004.

[KG99] Christoph Kern and Mark R. Greenstreet. Formal Verification In Hardware Design: A Survey. ACM Trans. Des. Autom.

Electron. Syst., 4(2):123–193, 1999.

[Kit95] Edward Kit. Software testing in the real world. Addison Wesley, 1995.

[NBAR04] Syed Nabi, Mahesh Balike, Jace Allen, and Kevin Rzemien. An overview of hardware-in-the-loop testing systems at visteon. SAE Technical Paper Series, 2004.

[Plfe01] Shari, Lawrence Pfleeger. Software Engineering: Theory and Practice, 2nd Edition. Prentice Hall, 2001.

[Pres05] Roger S. Pressman. Software Engineering: A Practitioner's Approach, Sixth Edition. McGraw Hill, 2005.

[Ray01] Erik T. Ray. Learning XML. O'Reilly, 2001.

[SAX06] Official Website for SAX. http://www.saxproject.org/. Visited date: Sept., 2006.

[SMH99] Holger Schlingloff, Oliver Meyer, Thomas Hülsing. Correctness Analysis of an Embedded Controller. DASIA '99, Lissabon (May 1999), 1999.

[Som04] Ian Sommerville. Software Engineering. Addison Wesley, Seventh edition, 2004.

[SSHF04] Matthias Stege, Frank Schäfer, Matthias Henker, Gerhard

Fettweis. Hardware in a Loop – A System Prototyping Platform for MIMO-Approaches. 2004 ITG Workshop on Smart Antennas.

[WBM97] S. Whorter, B. Baker, and G. Malan. Simulation system for control software validation. In SCS Simulation Multiconference, Alanta, 1997.

8. Appendix

Appendix A – Test Script

A typical test script generated by “TestCreator”:

```
<?xml version="1.0" encoding="UTF-8"?>
<test>
  <meta>

<result>E:\ep2\project\project\resources\testResults\testtextResult.xml</resu
lt>
    <description>this is a test</description>
    <name>Test</name>
  </meta>
  <componentList>
    <component class="Terminal">
      <namespace>http://www.eftpos2000.ch</namespace>

<templatePath>E:\ep2\project\project\resources\template\BEBackEnd\</tem
platePath>
      <encoding>0</encoding>
      <TrmID>chuan001</TrmID>
      <prefix>ep2</prefix>
      <interfaceName>BEBackEnd</interfaceName>
      <port>6625</port>
      <ip>127.0.0.1</ip>
```

```
      <name>Chuan Terminal</name>
    </component>
    <component class="PMS">
      <namespace>http://www.eftpos2000.ch</namespace>

<templatePath>E:\ep2\project\project\resources\template\BEBackEnd\</tem
platePath>
      <serverIP>127.0.0.1</serverIP>
      <encoding>0</encoding>
      <PMSID>999997</PMSID>
      <prefix>ep2</prefix>
      <interfaceName>BEBackEnd</interfaceName>
      <serverPort>6625</serverPort>
      <name>Fox PMS</name>
      <communicationMode>client</communicationMode>
    </component>
  </componentList>
  <testSequence>
    <event source="Fox PMS" target="Chuan Terminal">
      <conversation>
        <send>
          <PMSID>999997</PMSID>
          <type>posdeactack</type>
          <TrmID>01234567</TrmID>
          <msgnum>0123</msgnum>
          <filename>posdeactack.xml</filename>
        </send>
```

```
    </conversation>
    <conversation>
      <receive>
        <ActSeqCnt>99999994</ActSeqCnt>
        <PMSID>999997</PMSID>
        <POSID>123456</POSID>
        <type>posactntf</type>
        <TrmID>01348167</TrmID>
        <msgnum>0123</msgnum>
        <filename>posactntf.xml</filename>
        <ActDate>20010703</ActDate>
        <ActTime>134502</ActTime>
      </receive>
      <expected>
        <ActSeqCnt>99999994</ActSeqCnt>
        <PMSID>999997</PMSID>
        <POSID>123456</POSID>
        <type>posactntf</type>
        <TrmID>01348167</TrmID>
        <msgnum>0123</msgnum>
        <filename>posactntf.xml</filename>
        <ActDate>20010703</ActDate>
        <ActTime>134502</ActTime>
      </expected>
    </conversation>
  </event>
</testSequence>
```

</test>

Appendix B – Test Result

A typical test result script generated by “TestManager”:

```
<?xml version="1.0" encoding="UTF-8"?>
<testResult>
  <meta>

<result>E:\ep2\project\project\resources\testResults\testtextResult.xml</resu
lt>
    <description>this is a test</description>
    <name>Test</name>
  </meta>
  <componentList>
    <component class="Terminal">
      <namespace>http://www.eftpos2000.ch</namespace>

<templatePath>E:\ep2\project\project\resources\template\BEBackEnd\</tem
platePath>
      <encoding>0</encoding>
      <TrmID>chuan001</TrmID>
      <prefix>ep2</prefix>
      <interfaceName>BEBackEnd</interfaceName>
      <port>6625</port>
      <ip>127.0.0.1</ip>
```

<name>Chuan Terminal</name>
</component>
<component class="PMS">
<namespace>http://www.eftpos2000.ch</namespace>

<templatePath>E:\ep2\project\project\resources\template\BEBackEnd\</templatePath>
<serverIP>127.0.0.1</serverIP>
<encoding>0</encoding>
<PMSID>999997</PMSID>
<prefix>ep2</prefix>
<interfaceName>BEBackEnd</interfaceName>
<serverPort>6625</serverPort>
<name>Fox PMS</name>
<communicationMode>client</communicationMode>
</component>
</componentList>
<result>
<event source="Fox PMS" target="Chuan Terminal">
<conversation>
<send type=" posdeactack"><![CDATA[<?xml version="1.0" encoding="UTF-8"?>
<ep2:message xmlns:ep2="http://www.eftpos2000.ch">
<ep2:posdeactack msgnum="0123">
<ep2:PMSID>999997</ep2:PMSID>
<ep2:TrmID>chuan001</ep2:TrmID>
</ep2:posdeactack>

```
</ep2:message>]]></send>
      </conversation>
      <conversation>
        <expected type="        posactntf"><![CDATA[<?xml version="1.0"
encoding="UTF-8"?>
<ep2:message xmlns:ep2="http://www.eftpos2000.ch">
 <ep2:posactntf msgnum="0123">
  <ep2:ActDate>20010703</ep2:ActDate>
  <ep2:ActSeqCnt>99999994</ep2:ActSeqCnt>
  <ep2:ActTime>134502</ep2:ActTime>
  <ep2:PMSID>999997</ep2:PMSID>
  <ep2:POSID>123456</ep2:POSID>
  <ep2:TrmID>chuan001</ep2:TrmID>
 </ep2:posactntf>
</ep2:message>]]></expected>
      </conversation>
    </event>
  </result>
</testResult>
```

Appendix C – Implementation of BEBackEnd Interface

The complete code of the BEBackEnd interface:

```
package uk.ac.swan.ep2.xml;
import java.io.IOException;
import java.text.SimpleDateFormat;
```

```
import java.util.Date;

import org.jdom.JDOMException;

import uk.ac.swan.ep2.common.Config;
/**
 * Author        : Xu Ruiming
 * Filename          : XmlMessageFactory.java
 * Date          : Aug 1, 2006
 * Purpose : Handling messages between PMS and Terminal
 */
/**
 * @author Xu Ruiming
 * @since Aug 1, 2006
 * <p>
 * Handling messages between PMS and Terminal
 * </p>
*/
public class BEBackEnd extends XmlMessage {

    public BEBackEnd(Config config) throws IllegalArgumentException {
        super(config);
    }

    /**
     * Override super class getPrefixed template
     */
```

```
    public String getPrefixedTemplate() throws IOException,
JDOMException {
        String xmlMessage = super.getPrefixedTemplate();
        return xmlMessage;
    }
}
```

Appendix D – Implementation of Service Center

The code of the Service Center:

```
package uk.ac.swan.ep2.ep2Components;

import org.apache.log4j.Logger;

import java.net.ServerSocket;
import java.net.Socket;
import java.util.Iterator;

import uk.ac.swan.ep2.communication.network.Ep2Connection;
import uk.ac.swan.ep2.xml.Conversation;

/*
 * Author   : Xu Ruiming
 * Filename     : SericeCenter.java
 * Date      : Jul 25, 2006
 * Purpose: Performs service center related functions
```

```
 */

/**
 * @author Xu Ruiming
 * @since Jul 25, 2006
 * <p>
 * Service Center is able to run in two modes:
 * <li>Server</li>
 * <li>Client</li> <br/><br/>
 * Attribute <code>  communicationMode</code> in <code>  config</code> will determine
 * which mode the Service Center will run.<br/>
 * </p>
 */
public class ServiceCenter extends Ep2Component {
    /**
     * Logger for this class
     */
    private static final Logger logger = Logger.getLogger(ServiceCenter.class);

    /**
     * Setting a list of default setting in <code>config</code><br/>
     *
     */
    public ServiceCenter() {
        super();
```

```
        String[] key = { "prefix", "namespace", "class", "name", "serverIP",
                        "serverPort", "SCID", "interfaceName", "encoding",
                        "templatePath", "communicationMode" };
        String[] value = { "ep2", "http://www.eftpos2000.ch", "Acquirer",
"Fox SC", "127.0.0.1",
                        "6625", "00000000001", "SIConfig", "0",
                        "resources\\template\\SIConfig\\", "client" };

        // setting the default value
        for(int i = 0; i < key.length; i++) {
            config.set(key[i], value[i]);
        }
    }

    /**
     * Overriding base class <code>Ep2Component</code> <code>
run()</code> function
     */
    public void run() throws Exception {
        if(logger.isDebugEnabled()) {
            logger.debug("run() - start"); //$NON-NLS-1$
        }

        // Service Center operating in client mode
        if(config.get("communicationMode").equals("client")) {
            logger.info(config.get("name") + "(Service Center) Running in
Client Mode");
```

```
            clientMode();
        } else if(config.get("communicationMode").equals("server")) {
            logger.info("Service Center Running in Server Mode");
            serverMode();
        }

        if(logger.isDebugEnabled()) {
            logger.debug("run() - end"); //$NON-NLS-1$
        }
    }

    /**
     * Client Mode
     * @throws Exception
     */
    private void clientMode() throws Exception {
        if(logger.isDebugEnabled()) {
            logger.debug("clientMode() - start"); //$NON-NLS-1$
        }

        Socket socket;
        int port;

        try {
            port = Integer.parseInt(config.get("port"));
        } catch(NumberFormatException e) {
            logger.error("clientMode()", e); //$NON-NLS-1$
```

```
                logger.error("Client Invalid Port. Revert to default port:6625");
                port = 6625; // if the port is invalid, revert to default
            }
            socket = new Socket(config.get("ip"), port);
            logger.info("Connection Established at " + config.get("ip") + " port "
+ port);
            // run all the event
            runEvent(socket);

            if(logger.isDebugEnabled()) {
                logger.debug("clientMode() - end"); //$NON-NLS-1$
            }
        }

        /**
         * Server Mode
         * @throws Exception
         */
        private void serverMode() throws Exception {
            if(logger.isDebugEnabled()) {
                logger.debug("serverMode() - start"); //$NON-NLS-1$
            }

            ServerSocket serverSocket;
            Socket socket;
            int port;
```

```
    try {
        port = Integer.parseInt(config.get("serverPort"));
    } catch(NumberFormatException e) {
        logger.error("serverMode()", e); //$NON-NLS-1$

        logger.error("Invalid Port. Revert to default port:6625");
        port = 6625; // if the port is invalid, revert to default
    }

    serverSocket = new ServerSocket(port);
    logger.info("Service Center is starting with server mode at port "
            + port);
    // once client accepted the connection
    socket = serverSocket.accept();

    // run all the event
    runEvent(socket);

    // close the server socket
    serverSocket.close();

    if(logger.isDebugEnabled()) {
        logger.debug("serverMode() - end"); //$NON-NLS-1$
    }
}
```

```
    /**
     * Sending /receiving messages stored in <code>Server Center</code>
     * @param socket
     * @throws Exception
     */
    private void runEvent(Socket socket) throws Exception {
        if(logger.isDebugEnabled()) {
            logger.debug("runEvent(Socekt) - start"); //$NON-NLS-1$
        }

        Ep2Connection connection;
        Iterator it;
        // this number will be used through out all the messages
        int msgnum = 0;
        connection = new Ep2Connection(socket, config);

        connection.open();

        // playing all the conversation
        it = conversationList.iterator();
        logger.info(conversationList.size() + " conversation(s) for " +
config.get("name") + "(Server Center) to run");
        while(it.hasNext()) {
            Conversation con = (Conversation) it.next();
            con.setMsgnum(msgnum);
            con.run(connection);
```

```
            if(con.getMsgnum() > msgnum) {
                msgnum = con.getMsgnum();
            }
        }

        connection.close();

        if(logger.isDebugEnabled()) {
            logger.debug("runEvent(Socket) - end"); //$NON-NLS-1$
        }
    }
}
```