Implementation of a secure E-commerce solution for the Internet

Ward Vandewege

A thesis submitted in partial fulfilment of the requirements of the Katholieke Hogeschool Sint-Lieven, Department KIHO for the degree of Industrial Engineer,
carried out at the University of Central England.

June 1998

Katholieke Hogeschool Sint-Lieven, Department KIHO
and
University of Central England in Birmingham,
Department of Engineering and Computing Technology

Table of Contents

1. ABSTRACT....4

2. INTRODUCTION....5

2.1 Acknowledgements....5

2.2 Introduction....5

3. PROBLEM DEFINITION....6

3.1 Assignment....6

3.2 Details....6

3.3 Research....6

4. RESEARCH....7

4.1 Choices to make....7

4.2 OS to run the web server on....7

4.3 Web server....8

4.4 Encryption....9

4.4.1 Options....9

4.4.2 SSL....9

4.5 Programming language on the server....10

4.6 Programming language on the client....12

4.7 Databases....13

5. SERVLETS AND NES....14

5.1 Running servlets under NES....14

5.2 Calling servlets....14

5.3 The basic structure of a servlet....16

6. SPECIFICATIONS....19

6.1 Two interfaces....19

6.2 Customer interface....19

6.3 Administration interface....20

7. DESIGN....22

7.1 Preamble....22

7.2 Multi-threading....22

7.3 The secure connection....22

7.4 Modules....23

7.3.1 Product data class....23

7.3.2 Customer data class....23

7.3.3 Transaction data class....24

7.3.4 Products servlet class......24
7.3.5 Users servlet class......24
7.3.6 ProcessTransactions servlet class......25
7.3.7 Serve servlet class......25
7.3.8 ServeParser class......27
7.3.9 Management servlet class......27
7.3.10 Cart servlet class......28

8. IMPLEMENTATION......29

8.1 PREAMBLE......29
8.2 THE SOURCE CODE EXPLAINED......29
8.2.1 Product.java......29
8.2.2 Customer.java......31
8.2.3 Transaction.java......33
8.2.4 Products.java......34
8.2.5 Users.java......42
8.2.6 ProcessTransactions.java......51
8.2.7 Serve.java......59
8.2.8 ServeParser.java......63
8.2.9 Management.java......66
8.2.10 Cart.java......71
8.2.11 CheckCC.java......76
8.2.12 Sorter.java......76
8.2.13 CustomerDetailsFormCheck.js......76

9. INSTALLING THE SOFTWARE ON A COMPUTER......78

10. CONCLUSION AND FURTHER WORK......79

11. APPENDICES......80

11.1 REVIEWS AND DOCUMENTATION......80
11.1.1 INSTALLING NOVELL NETWARE 4.11 SERVER......80
11.1.2 INSTALLING WINDOWS NT 4.0 OVER A LAN......82
11.1.3 CONFIGURING NETSCAPE ENTERPRISE SERVER 3.5.1......83
11.1.4 JBUILDER 1.1......92
11.2 SOURCE CODE......94
11.3 GENERAL.CSS......94
11.4 USED ABBREVIATIONS......95

12. REFERENCES......96

1. Abstract

The assignment was to implement a secure E-commerce solution for the Internet, in an isolated LAN consisting of 3 machines: a Novell-server for authentication on the LAN, a NT-server to run the web server and a NT-workstation with Netscape Communicator to simulate the users.

After some research, I have chosen to use Netscape Enterprise Server (NES) 3.5.1 as web server and Java Servlets to do all information processing on the server. Javascript is used to add some intelligence on the client-side. Also on the client-side, cookies are used to keep track of temporary data. The security of purchase is guaranteed by the use of the Secure Sockets Layer (SSL) protocol.

Practically, this thesis project consists of a web site that allows the secure purchase of items over the Internet, featuring shopping carts, Credit Card validation, a database of customers and a database of products. It also contains documentation/reviews on: JBuilder 1.1, Installing Netware 4.11, Installing Windows NT over a LAN and configuring Netscape Enterprise Server 3.5.1.

2. Introduction

2.1 Acknowledgements

First of all I want to thank my supervisor at the UCE, Chris Noble, and at the KIHO Dr. Luc De Backer, for the great support. Furthermore many thanks to Dr. Simon Handley of the UCE for taking care of me and the other Socrates students. Many thanks also to Richard Kay for the enriching exchanges of ideas. And a lot of thanks to Dean England and John Higgins for helping me out with many practical issues. Thanks also to Imran Yousaf. And finally a big thank you to my parents and (international) friends.

2.2 Introduction

Over the recent years, trading on the Internet has only just begun. *eMarketer* (http://www.emarketer.com), a New York based market research firm, estimates that consumers will spend $4.5 billion buying goods on the Internet in 1998. In 1997, electronic commerce was good for 1.8 billion dollar, and by 2002 they estimate it to reach $26 billion, almost a six-fold of the estimated 1998 figure.

The real big numbers lie in *Business to business* electronic commerce, however. *eMarketer* estimates that sales in this segment will rise form $5.6 billion in 1997 to nearly $16 billion in 1998 and an amazing $268 billion in 2002.

All these number have to be put in perspective though; for instance, *eMarketer* predicts that the online sale of airline-tickets will increase from 1% of the total sales today to 5% in 2002. This is still only a small share, but looking at the pace E-commerce is gaining importance with, it is not hard to imagine that within 10 to 15 years, E-commerce will account for a major share of the total world trade.

For obvious reasons, all this trade needs a secure channel to encrypt sensitive data like credit card numbers. And that is why this thesis project focuses on the development of a secure E-commerce solution for the Internet.

3. Problem definition

3.1 Assignment

The assignment consists of the implementation of a secure E-commerce solution for the Internet, in an isolated LAN consisting of 3 machines: a Novell-server for authentication on the LAN, a NT-server to run the web server and a NT-workstation with Netscape Communicator to simulate the users. Some research is necessary to decide which web server to use, what technology to use for the information processing on the server, etc.
Practically, the goal of this project is the development of a web site that allows the secure purchase of documents over the Internet.

3.2 Details

The proposed setup is drawn in fig. 3.1:

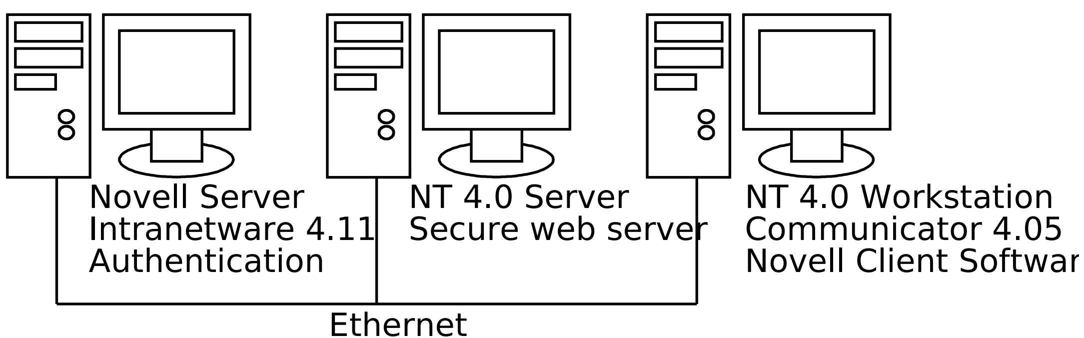

Fig. 3.1 Proposed setup

Novell Server: P166MMX, 32MB RAM
NT 4.0 Server: P166MMX, 32MB RAM
NT 4.0 Workstation: P166MMX, 32MB RAM

This setup with the Novell server for authentication on the LAN has been chosen to allow easy merger with the existing UCE-DECT network later on. In this network, all authentication is being handled by a Novell server.

3.3 Research

The next chapter of this report looks into what solutions are currently available, and makes a brief analysis of the used techniques, price,... of the various products on the market.

In particular I look into a company named Novonyx, a joint-venture of Novell and Netscape, that is porting the various Netscape Server products to the Novell platform. I find out if it is worth it to drop the NT server and use the Novell machine for the (secure) web server as well.

4. Research

4.1 Choices to make

In the following paragraphs, I will take a look at the options that exist when developing an E-commerce solution. The main choices to make are: what OS to run the web server on, which web server to use, what encryption protocol to use, and which programming or scripting languages to use on the server and on the client.
A final thing to consider is which commercial database to use.

4.2 OS to run the web server on

Obviously, the choice of the OS to run the web server on depends mainly on the choice of the web server, because most web servers are only available for a few Operating Systems. Another consideration is the availability of Operating Systems at the UCE. Licences are available for Novell Netware 4.11 and NT Sever 4.0. So this narrows the choice down to these 2 Operating Systems and Linux, a - among academics - popular freeware POSIX compliant UNIX-implementation for Intel, PowerPC, Alpha,... . The third criterion is the machine this OS was to run on, in this case an Intel-processor based PC. The three Operating Systems mentioned above are available for the Intel platform.
The E-commerce solution I set up during this thesis will be used as a demonstrator for DECT-students in the future, and that is why my supervisor prefers NT or Netware over Linux, because the importance of the latter OS outside the academic world is rather limited. It is therefore not a prominent part of the education in the Department of Engineering and Computer Technology at the UCE.

4.3 Web server

There are many different web servers available on the market today. Some are free, but most are not. The most popular web servers today on the Intel-platform are:

Product	Company	Runs on ...	Price
Apache	The Apache Group	Different flavours of UNIX	Free
HTTPd	NCSA	Different flavours of UNIX	Free - no longer supported
IIS 4.0	Microsoft	NT	Free with NT Server 4.0
NES 3.5.1	Netscape	NT and Sun Solaris	Free for educational institutions, otherwise 1295 $
NES 3.0	Novonyx	Novell Netware 4.11	Not free; see http://www.novonyx.com
Lotus Domino	Lotus	NT, OS/390, Windows 95, OS/2 Warp, AIX, Sun Solaris, HP-UX	free 30-day trial; after that free with SSL disabled; otherwise contact http://www.lotus.com
JavaServer 1.1	Sun	NT and Sun Solaris	free 30-day trial; after that 295 $
...			

Because the desired web server must should on NT or Netware, the UNIX-web servers Apache and HTTPd are ruled out.
NES is available on the 2 platforms, NT and Netware. The Netware version however is not free for educational purposes, and, more importantly, it is not the latest available version (3.0 as opposed to 3.5.1 on NT). This ruled out Novonyx' NES for me.
Because Novonyx' product is the only one available for Netware, our OS of choice will consequently be NT Server 4.0. A free web server is obviously preferred, so IIS and NES 3.5.1 are preferred over Lotus Domino and JavaServer.
I have used IIS 3.0 before, and, when I started this thesis project, this was still the latest available version. It has some advantages, but I dislike the poor documentation and the horrible file-permissions. Instead of using Access Control Lists (ACL) as most other web servers do, IIS 3.0 security relies entirely on setting the correct file permissions in the OS, a tedious process. This made me a bit reluctant to use IIS, and there is also the point of not getting too reliant on one software vendor. Indeed, NT is a Microsoft OS, and IIS is a Microsoft product as well.
At the end of march IIS 4.0 was released, but by then I had made the decision to use Netscape Enterprise Server 3.5.1 on NT, and my project had advanced too far to change web servers.

4.4 Encryption

4.4.1 Options

There are a few proposed encryption and user authentication standards for the Web. The most well known are Secure Sockets Layer (SSL) and Secure HyperText Transport Protocol (SHTTP). Each requires the right combination of compatible browser and server to operate. The last few years SSL, the scheme proposed by Netscape, has become the de-facto standard, as it is the only protocol supported by the two major browsers available, Netscape Communicator and MS Internet Explorer (IE). The SSL protocol is implemented in most web servers, including NES and IIS. This makes SSL a good choice for this thesis project. The original SSL proposal by Netscape can be found at http://home.netscape.com/newsref/std/SSL.html.
SHTTP has only been implemented in the Open Market Server marketed by Open Market, Inc. on the server side, and in Secure HTTP Mosaic by Enterprise Integration Technologies on the client side. This makes SHTTP not a good choice for this thesis project - obviously there is no real off-the-shelf alternative today for SSL.

4.4.2 SSL

SSL currently uses 128-bit encryption in the USA and 40-bit encryption outside, because of American export regulations. Actually, outside the USA, 128-bit encryption is used as well, but only 40 of these 128 bits are hidden. Secure Sockets Layer (SSL) is a protocol situated in the network layer, above TCP-IP, but below HTTP, FTP,... It uses a combined symmetric/public key encryption approach. Symmetric encryption uses one key for both encryption and decryption.

Public key encryption:

Public key encryption uses 2 keys, one for encrypting and one for decrypting. Public key encryption is easy to calculate in one way (encryption) but very hard to calculate the other way (decryption). The keys are called the public key and the private key. You can distribute your public key freely, but you may ***never*** reveal your private key to anyone. A message encrypted with your public key can only be decrypted with your private key, and vice versa. So if someone wants to send you a message that only you can decrypt, he (or she) encrypts the message with your public key. This means it can only be decrypted with your private key, which only you have. But it works the other way around as well. Imagine you want to publish a message to the world, but so that everyone can be sure it originated from you. You would then "sign" the message using your private key, i.e. you would add a block encrypted with your private key to the unencrypted, readable message. This encrypted block contains a digital signature of the original document you sent out. Since the encrypted block can only be decrypted using your public key, everyone can see it originated from you. And they can also

see if the document has been tampered with, because the digital signature in the encrypted block must match the accompanying document.

The encryption of SSL:

Because asymmetric (public key) encryption is slower than symmetric encryption, SSL uses a combination of the two. When a server and a client start a new connection, they start by exchanging their public keys. Then the client calculates a symmetric key for this session, encrypts it with the servers public key and sends it to the server. The server decrypts the message using its private key, and from that point on all encryption for this session is done using the - faster - symmetric key.

The "strength" of a key is measured by the number of bits used in certain numbers creating the key. In the US, SSL uses 128 bit encryption, which is - now - very hard to break. But the international version of SSL only uses 40 bit encryption (due to US export regulations), which isn't all that secure. But when considering encryption, the most important thing is the value of the information versus the cost or time to crack the encryption. Eventually, the American Export regulations will change, and when that happens, no changes will be necessary to this thesis project, because I just use standard SSL. When running this program with a new server and client that support SSL with more bits, the information that is sent using SSL will be more secure, without changes to my source code.

How does the encryption work?

The encryption keys discussed higher are complex mathematical functions that are easy to compute in one direction, but very hard to compute in the other direction. Because of this, it is very hard for someone to decipher your private key or your message even when he has your public key and the encrypted message. The types of functions used in public key encryption are very complex and resistant to pattern searches because they use prime numbers in their calculations. There are no patterns for determining prime numbers, so examining the cyphertext for patterns won't do much good.

But if decrypting an encrypted message is so hard, how come with the private key it is quickly possible? The private key is also a complex mathematical function, embedded in the public key as a shortcut to "solving" the public key function (and thus decrypting the encrypted message). It is as difficult to determine the private key from a public key as it is to decrypt an encrypted message.

4.5 Programming language on the server

The Common Gateway Interface (CGI) is located on the server, and it is an interface between programs and a HTTP server. Like a door between two rooms, the CGI is between programs and the HTTP server. CGI lets these programs access information coming from the client (such as HTML forms, data,...) and send a response back to the browser of the client. This response can be anything that the browser of the client understands (HTML, plain

text, sound, video,...). This allows web-pages to become interactive, to behave differently depending on user-input. A typical use for CGI-programs or scripts, as the programs that are accessible through the HTTP server are commonly called, is accessing a database of some kind.

CGI-programs used to be written mainly in Perl and C, but this is changing now with the advent of Java and Javascript. Netscape is pushing Javascript as the development platform for NES, but Javascript has the disadvantage that it is only a scripting language. This means that it is relatively easy to learn and use, but also that it is limited in its features. This was the primary reason I did not choose Javascript as the programming language on the server.

Instead I chose to use Java Servlets, a new technology that allows powerful CGI-programming in Java. Servlets are Java classes loaded into and invoked by a Web server. In fact, they are the server equivalent of applets on the browser side, hence the name servlets.

The big advantages that Java has over traditional programming languages, such as C(++), are:

- Portability: Java source code and compiled byte code work unchanged across a multitude of platforms, including Windows, MAC OS and most flavours of UNIX.
- Features: Java has plenty of powerful features such as built-in threads, exceptions, security and an excellent Object Model,...
- Ease of use: despite all these features, Java is a lot easier to use and simpler than C++.
- Network-aware: Java is inherently network-aware, with objects that map to sockets (TCP/IP network connections), URLs, ...
- ...

Java Servlets have additional advantages over traditional CGI programming: performance and reliability. When using typical C or Perl CGI-solutions, for every incoming CGI request the CGI program loads, initialises, executes, and finally returns information (HTML,...). This is a time-consuming process, and many simultaneous CGI requests can quickly bring a high-traffic server to its knees.

Both Microsoft and Netscape have tried to do something about this performance issue, by releasing APIs to allow programmers to write CGI-programs as libraries, that load as part of the server itself. This greatly improves performance because the libraries are loaded when the server starts, and not killed after every request. But this approach also places more responsibility on the programmer: a badly written library can easily crash the complete Web server. This seriously endangers the reliability of the Web server.

Java Servlets provide solutions to the problems of performance and reliability. Once a Java Servlet has been invoked (not necessarily when the server starts; more likely when it is called the first time), it stays alive and ready for the next request. When multiple requests are made at the same time, more instances of the servlet are created - provided you did not specify the single thread model for your Servlet, in which case all requests will be processed sequentially.

Because Java is pointerless, and has an excellent garbage-collector, which takes away the complex memory-management task from the programmer, a Java Servlet is much less likely to misbehave sufficiently to crash the server than a C++ CGI.
Additionally, you can use the full power of the Java language when developing Java Servlets. Java has numerous standard libraries for useful features like accessing ZIP-files, ODBC database access through Java Database Connectivity (JDBC), object serialisation,...
There is one downside to Java in general and Java Servlets in particular. It is a young technology, and therefore it is constantly evolving. I will describe the problems I ran into because of this later in this report.

4.6 Programming language on the client

Because we do not live in a perfect world, not everyone has a T1 connection to the Internet. Consequently, as any web site, a good e-commerce solution should be economical with the transport of information over the Internet, to reduce the waiting time for the customers. One of the ways to do this, is to add some intelligence on the client side, in the browser of the potential customer.
When the customer fills out a form, certain fields have to comply with rules, e.g., his or her name should not be empty and the Credit Card number he or she specifies should be valid. If this information has to be sent to the server to be verified there, and then returned to the client if it is not correct, a lot of overhead is introduced. This overhead can be avoided by adding some sort of form-verification in the browser on the client-side. There are not many alternatives to achieve this goal - only Javascript and VBscript. Because Javascript is currently the only one supported on both major browsers, my choice was quickly made in favour of it.

4.7 Databases

An E-commerce site obviously needs a database. In our case, we need three databases: one for the products, one for the customers, and one for the transactions.

There were no heavy-duty databases like Oracle, SyBase or IBM's DB2 available to use at the UCE/DECT. The only databases available were MS Access, Paradox or Inprise's (formerly Borland) InterBase. Of these, Access and Paradox are not heavy duty - some people refer to them as "toys".

There is another option: not to use a database, but simply save the information in a proprietary format. The disadvantage of this approach is that the E-commerce solution will not be able to support thousands of users, products or transactions, simply because it will become too slow when big quantities of data are involved. But the big advantage is that the solution stays very portable, because all data access is done in Java and saved directly in files. Choosing for an Access or Paradox database - accessible through our servlet via JDBC - would bind us to Windows servers. This seems a big sacrifice to make, especially because using these "toy"-databases will not allow to make a real industrial strength e-commerce solution anyway.

So I chose to use no commercial database, but instead save all information directly in files on the server. This obviously limits the solution in scalability - I think it will be too slow for more than a few hundred transactions a day, a few dozens products and a few hundred users - though much will depend on the hardware. If more than that would be required, a high-end server hardware would be necessary anyway - though switching to Linux might improve performance considerably with the same hardware.

5. Servlets and NES

5.1 Running servlets under NES

Currently (April 1998) the last stable Java Development Kit (JDK) version is 1.1.6, but this version is not yet supported in many applications. Netscape Communicator for instance, only supports Java 1.1 natively since version 4.05, which came out in April 1998. Because the Java Servlets are executed on the server, what version of Java the client's browser supports is irrelevant, but I ran into similar compatibility issues with NES. Netscape claims that Enterprise Server 3.5.1 supports JDK 1.1.x, but I had compatibility problems when running my servlets. Using the Servletrunner provided with the Java Servlet Development Kit 1.1.5 (the last standalone version; from JDK version 1.2 the JSDK is included with the JDK) they ran without problems, but when I tried to run them under NES they crashed continuously.

The exact cause of the crashes was unclear, but had something to do with the rather limited implementation of Servlets in NES 3.5.1.

There are solutions to this. Several companies offer "plugins" for different web-servers. These plugins are more powerful implementations of Java Servlets than the one provided with the server, in this case NES. After some research, I came across some laudative comments on such a plugin called JRun, by Live Software. I decided to try it, and after a flawless installation ... my servlets didn't run at all any more.

The problem was the version of the JSDK. JRun 2.1.2, the version I am using, uses JDK 1.2 Beta 3, which includes the 1.2 (Beta)version of the JSDK. And there are some significant changes in the Servlet API between version 1.1.5 and 1.2, since the four servlet packages that existed in the former are now merged in two packages, javax.servlet and javax.servlet.http. So after the porting of the code I had written so far, my servlets ran without problems. And the best thing about JRun is that it is free.

5.2 Calling servlets

When the web-server loads a servlet, it refers to a file *servlets.properties*, which resides in one of the configuration directories of the server. For information on where to find this file in NES, refer to appendix 11.1.3, point 14, *using Java Servlets in NES*. When JRun is installed, the *servlets.properties* file usually resides in the \Program Files\Live Software\JRunNSAPI\servlets\properties\ directory. The *servlets.properties* file as used in this project can be found in table 5.1.

```
#
# Servlets Properties
#
# servlet.<servlet name>.code=class name (foo, not foo.class)
# servlet.<servlet name>.args=list of {name, value} pairs which can be accessed
#                         by the servlet using the servlet API calls
# servlet.<servlet name>.preload='true' or 'false' determines whether this servlet
#                         is loaded at server startup
servlet.jrunssi.preload=true
servlet.jrunssi.args=
servlet.jrunssi.code=com.livesoftware.jrun.plugins.ssi.JRunSSI
#Added by Ward Vandewege
servlet.Products.preload=false
servlet.Products.code=Products
servlet.Products.args=filename=E:/temp/products.dat
servlet.Products.initArgs=filename=E:/temp/products.dat
servlet.ProductsDebug.preload=false
servlet.ProductsDebug.code=Products
servlet.ProductsDebug.args=filename=E:/temp/products.dat,debug=true
servlet.ProductsDebug.initArgs=filename=E:/temp/products.dat,debug=true
servlet.Users.preload=false
servlet.Users.code=Users
servlet.Users.args=filename=E:/temp/customers.dat
servlet.Users.initArgs=filename=E:/temp/customers.dat
servlet.UsersDebug.preload=false
servlet.UsersDebug.code=Users
servlet.UsersDebug.initArgs=filename=E:/temp/customers.dat,debug=true
servlet.UsersDebug.args=filename=E:/temp/customers.dat,debug=true
servlet.Cart.preload=false
servlet.Cart.code=Cart
servlet.Cart.args=debug=false
servlet.Cart.initArgs=debug=false
servlet.CartDebug.preload=false
servlet.CartDebug.code=Cart
servlet.CartDebug.initArgs=debug=true
servlet.CartDebug.args=debug=true
servlet.ProcessTransactions.preload=false
servlet.ProcessTransactions.code=ProcessTransactions
servlet.ProcessTransactions.args=filename=E:/temp/transactions.dat,debug=false
servlet.ProcessTransactions.initArgs=filename=E:/temp/transactions.dat,debug=false
servlet.Serve.preload=false
servlet.Serve.code=Serve
servlet.Serve.args=debug=false
servlet.Serve.initArgs=debug=false
servlet.ServeDebug.preload=false
servlet.ServeDebug.code=Serve
servlet.ServeDebug.args=debug=true
servlet.ServeDebug.initArgs=debug=true
```

Table 5.1
Servlets.properties file

Every servlet has 2 sets of lines; one set with lines of the form *servlet.servletname....*, and one with lines of the form *servlet.servletnameDebug....* Each set has 4 lines; three of which are necessary for JRun. The first line, *servlet.servletname.preload* determines if the servlet should be preloaded when the server starts. If this is true, the servlet will be loaded at server setup; otherwise, it will be loaded when it gets first called. The second line, *servlet.servletname.code* determines which servlet should be called when *servletname* appears in a URL. The extension *.java* should not be specified. The next line, *servlet.servletname.initArgs,* lists parameters that are passed to the servlet. The parameters should be specified in the form *paramname=value,paramname=value,...*

So, putting this all together, when the URL *http://some.server.name/Servlet/ProductsDebug* is requested, the servlet *Products* gets called because of the line "servlet.ProductsDebug.code=Products" in the *servlets.properties* file, with the arguments as specified in "servlet.ProductsDebug.initArgs=filename=E:/temp/products.dat,debug=true". So in this case, the *debug* parameter is set to *true.* When the URL *http://some.server.name/Servlet/Products* is requested, the same servlet is called because of the "servlet.Products.code=Products" line, but this time with the debug parameter set to *false*

because it is initialised differently in the *initArgs* line. This is an easy way to toggle the debug output on or off, because all the servlets set the class parameter *debug*, according to the value passed to them from the *servlets.properties* file. This parameter is checked every time debug output is written.

The fourth line, *servlet.servletname.args* has the same function as the *initArgs* line, but for some unclear reason the JRun people have changed the ServletRunner default (*args*) to *initArgs.* I include both lines to allow compatibility with the ServletRunner, as provided with the JDK. The *initArgs* line is ignored when the ServletRunner uses the file, and the *args* line is ignored by JRun.

5.3 The basic structure of a servlet

A web-servlet is a descendant of the HttpServlet class, which adds HTTP-specific methods to the generic servlet interface. Servlets usually override the superclass methods *init(ServletConfig config)*, *destroy()*, *doPost(HttpServletRequest req, HttpServletResponse res)*, *doGet(HttpServletRequest req, HttpServletResponse res)* and *getServletInfo()*.

The method *init* is called the first time when the servlet is loaded, and could for instance be used to connect to an external database using JDBC, or to open a file. If you define *init* in your servlet, you are overriding the *init* method in superclass HttpServlet. The *init* method in HttpServlet does some important initialisation, so you must call *super.init(config)* from somewhere in your version of *init*. Just before the servlet is destroyed, *destroy()* is called. It could e.g. be used to close the connection to a database, or to close any open files.

The methods *doPost* and *doGet* are invoked when the Web server gets a Post respectively Get request from the client. In both cases, two parameters are passed in: *HttpServletRequest req* and *HttpServletResponse res*. The HttpServletRequest is an object that contains all the information about the request (e.g. the client's IP address, host name, and request parameters). The parameter *res*, of the type HttpServletResponse, is an object that allows our servlet to respond to the request. Using *res*, we can set the type of data we are returning (e.g. text or html), find the outputstream we should write to, and - in our case - set Http-specific headers.

You should also override *getServletInfo().* This method should return a short description of the servlet, that then can be displayed in a server's administration interface.

The following example is a very basic servlet that only overrides the methods *doPost*, *doGet* and *getServletInfo*. Because there is nothing to do when the servlet loads or is destroyed, *init* and *destroy* are not implemented, resulting in calls to the superclass's corresponding methods when the server loads or destroys the servlet.

This sample servlet shown in table 5.2 counts the number of hits from the originating IP address.

```
import java.io.*;
import java.util.*;

import javax.servlet.*;
import javax.servlet.http.*;

/**
 * Servlet IPCount
 * This sample servlet counts the number of hits with the originating
 * IP-address.
 *
 * @version          1.00, 25/05/98
 * @author           Ward Vandewege (wardv@usa.net)
 */
public class IPCount extends HttpServlet {
  Hashtable countHash = new Hashtable();

  /**
   * Called by the web server when a HTTP POST request is made.
   * @param req The POST request information
   * @param res The HTTP response object
   */
  public void doPost (HttpServletRequest req, HttpServletResponse res)
            throws ServletException, IOException {
    //value chosen to limit denial of service
    if (req.getContentLength() > 8*1024) {
      res.setContentType("text/html");
      ServletOutputStream out = res.getOutputStream();
      out.println("<html><head><title>Too big</title></head>");
      out.println("<body><h1>Error - content length &gt;8k not allowed");
      out.println("</h1></body></html>");
    } else {
      processRequest(req,res);
    }
  }
```

```
  /**
   * Called by the web server when a HTTP GET request is made.
   * @param req The GET request information
   * @param res The HTTP response object
   */
  public void doGet (HttpServletRequest req, HttpServletResponse res)
              throws ServletException, IOException {
    processRequest(req,res);
  }

  /**
   * Process the requests coming from doPost and doGet
   * @param req The POST/GET request information
   * @param res The HTTP response object
   */
  private void processRequest(HttpServletRequest req, HttpServletResponse res)
             throws ServletException, IOException {
    String remoteAddress = req.getRemoteAddr();
    Integer hits;
    synchronized(countHash) {
      if (countHash.containsKey(remoteAddress)) {
        hits = (Integer)countHash.get(remoteAddress);
        hits = new Integer(hits.intValue() + 1);
      } else {
        hits = new Integer(1);
      }
      countHash.put(remoteAddress, hits);
    }
    res.setContentType("text/plain");
    ServletOutputStream out = res.getOutputStream();
    out.println("This site has been accessed "+ hits +" time"+((hits.intValue() > 1) ? "s" : "")+
                  " from your IP-address: "+ remoteAddress);
  }

  /**
   * getServletInfo() returns a short description of a servlet.
   */
  public String getServletInfo() {
    return "Returns number of hits by IP address.";
  }
}
```

Table 5.2: a simple servlet

6. Specifications

6.1 Two interfaces

When developing an E-commerce web site, there are 2 interfaces to be created. The first one is the customer interface, where items can be purchased and searched for, and where personal information can be updated. But an administration interface is also necessary. Here, a site administrator should be able to manage products, users and transactions.
In this case, both of these interfaces are accessible through a web browser, thus allowing easy access to the web site from all over the world, also for the administrator. The big advantage of the browser-approach is that there is no need to create a proprietary front-end for the administration of the site. The administrator can do his job from whatever machine he wants, as long as it has a web-browser. And web-browsers are universally available for almost every platform.

6.2 Customer interface

The customer interface is what people will interact with when they visit the site. Consequently it should be easy to use, self-explaining and straightforward. The necessary features are: a product list, a product search facility, an easy way to order products and a means to update the customers' personal information.
The easy way to order products needs some further thoughts. In real life, when we go shopping, we don't take one item, pay for it, take another one, pay for it, etc. No, we collect the products we need, and pay for them all together. To collect the items we need, we often use a shopping cart. Hence, it would be nice to implement the equivalent of a shopping cart for the E-commerce site. This implies the implementation of extra features: viewing and editing the shopping cart.
The site will sell documents. There are 2 options to do this: either a download is allowed after purchase, or the user is simply allowed access to the online documents. The first approach has two disadvantages. First, it is not unlikely that something goes wrong during the download: transmission faults, aborted downloads,... are very common today on the Internet. So we might end up charging customers for documents that they did not receive properly, probably resulting in a lot of complaints. Secondly, once the user has downloaded documents, he or she has no easy access to corrected versions of the documents, updates,...
These two disadvantages made me adopt the second solution. Once the transaction is saved, the user can login to a personalised page where the documents he or she purchased access to are listed. These documents can be saved locally if the user wishes to, but the most recent version will always be available online. This is especially interesting for online courses. The (simplified) customer site map is shown in figure 6.1.

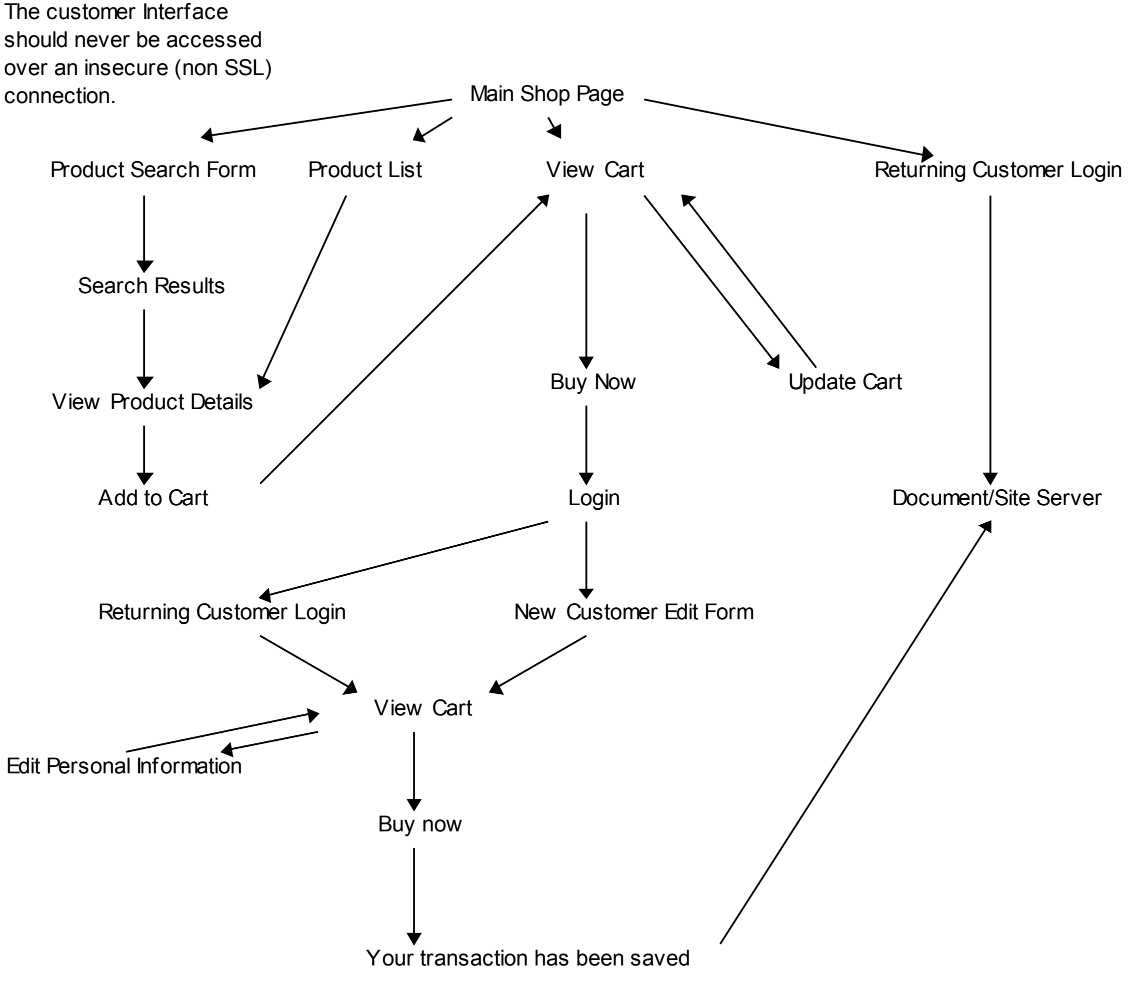

Fig 6.1 Customer site map

6.3 Administration interface

The administrator of the site is the only person who will use the administration interface. He or she can be expected to be knowledgeable about E-commerce and the managing of such a site. As a result, the stress should rather be on powerful features than on ease of use and straightforwardness - not entirely forgetting the latter two of course.

First of all, the administrator needs to provide his credentials (i.e. his login name and password). To secure the login information and the rest of the information that is possibly sent over the Internet when the administrator is working, the administration interface should be accessed only over a secure (SSL) connection. Ideally, the administration interface would notice if somebody tried to login over a non-secure connection and warn about it, or even deny access.

The necessary features are: product management, user management and transaction management. Product management should include listing, searching, editing and adding of products. The searching should be available on as much fields as possible.

User management should include listing, searching, editing and adding of users - though the latter will usually not be necessary from the management interface, as this happens through

the user interface. Searching users needs only to be available on user name and credit card number.

Transaction management should include listing, searching, editing and processing of transactions. Searching of transactions should be available on the product and user name field. Processing of transactions is what the administration interface will be used for most of all, so special attention should be paid to ease of use and straightforwardness of this feature. Best would be that the administrator would be provided with a list of unprocessed transactions, which he could cycle trough easily, marking a “processed” field on each as he processes it (i.e. charges the credit card). The management site map his is shown in figure 6.2.

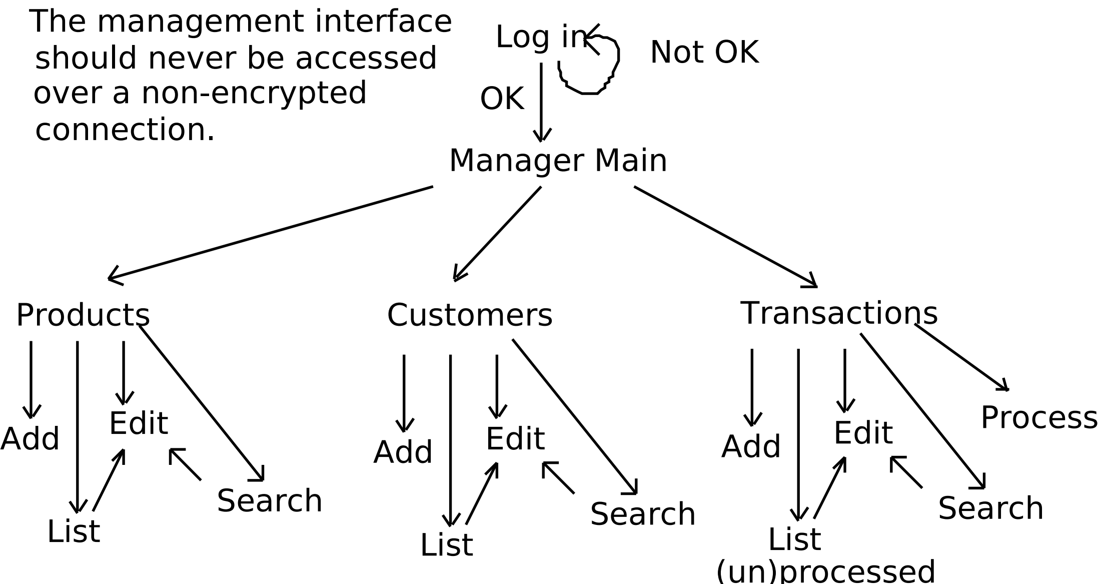

Fig.6.2 Management Site Map

7. Design

7.1 Preamble

In this chapter, the design of the software is discussed without going into system-specific properties of the software. The actions the software should undertake when a user request a product list, buys something etc. are discussed.

7.2 Multi-threading

As discussed higher, the software consists of 2 parts: a user interface and an administration interface. This division of tasks may seem obvious from an outside point of view, but I chose for another approach. I chose to develop three servlet classes that do all the interfacing with their respective database (i.e. Products, Customers and Transactions). These databases - hashtables - are *private*, which means they can only be accessed directly from within the class. Other classes can only access the hashtables through the methods of their respective class. I thought this approach would be the easiest way to keep the data integer, something that requires special attention because Java is multi-threaded.

Multi-threading means that there can be multiple threads or instances of a class running at the same time. This makes it possible for multiple people to buy things at the site at the same time, without problems - provided the structures that are critical are synchronised. It must be avoided at all cost that 2 threads access a structure, e.g. a hashtable, and try to alter it at the same time. Therefore the java keyword *synchronized* can be used. You can synchronise on a variable, thus allowing only one thread at a time access to this variable in a particular piece of code, or you can put the *synchronized* keyword in a method declaration, thereby only allowing access to the method for one thread at a time.

7.3 The secure connection

The approach with the three servlets that do all data interfacing has another consequence, regarding SSL. Usually, e-commerce sites work over a normal http connection, and switch to a *secure server* for the critical parts of the site, e.g. when the customer has to enter his credit card details. This *secure server* is not necessarily another machine, but more likely just another webserver, a secure one, running on the same machine. This was also the case in my development environment.

The hashtables I use, *products*, *customers* and *transactions*, are all *static*, which means that there is only one copy loaded in memory, and it is associated with the respective class itself, not with the instances of the class. All instances can access it, but there is only one copy. This is obviously necessary for data integrity. The hashtables are read from file when the first instance of its class is loaded. When changes are made to the hashtable, the whole hashtable is re-written to file.

At first I was intending to make the whole customer interface non-secure, and only change to a secure connection for the editing of the personal details of the customer. The administrator interface was going to be accessed only over a secure connection. But this approach proved impossible.
The problem is that the secure and the normal web-server are completely separated. That means that when the secure server is invoked, it loads a new copy of the classes, reading the hashtable files from disk. Then, when a customer edits his details and saves them, the hashtable in the memory of the secure server is updated, and written to disk - to the same file as the normal server uses. Because the normal server does not reload the file every time it has to look up something in the hashtable (for performance reasons), it never sees the changes made to the file on disk until it is restarted. Even worse, if some change is made to the hashtable in the memory of the normal server, it writes a copy of its hashtable to disk, thus overwriting all changes made in the secure server. This is clearly unacceptable, and that is why I decided to run the whole site over a secure connection. It is a bit slower, but it is more secure and the only solution to the problem described above given the design of this application.
The origin of the problem lies of course in the fact that I don't use an external "industrial strength" database - for reasons explained in section 4.7.

7.4 Modules

This modular approach is important. It implies that the three servlets that interface with a database must provide all the methods that are needed, e.g. for adding, editing and deleting of users. This means also that if someone later would like to use an industrial strength database instead of the hashtables I use, he should simply replace those three classes with classes that interface with the external database and provide all the functionality that the ones I wrote provide.

7.3.1 Product data class

The product data class should contain all the fields with necessary information about a product. The class should certainly define a unique *id* field that can be used to identify a product, and to retrieve a product from the database of products using the unique *id* field as the hashtable key. A method should also be defined to initialise the *id* field for new products.

7.3.2 Customer data class

The customer data class should contain all the fields with necessary information about a customer. The class should also define a unique *id* field that can be used to identify a customer, and to retrieve a customer from the database of customers, using the unique *id* field as the hashtable key. Here, too, a method should also be defined to initialise the *id* field for new customers.

7.3.3 Transaction data class

The transaction data class should contain all the fields with necessary information about a transaction. The class should define a unique *id* field that can be used to identify a transaction, and to retrieve a transaction from the database of transactions, using the unique *id* field as the hashtable key. A method should also be defined to initialise the *id* field for new transactions.

7.3.4 Products servlet class

The products servlet class should do all interfacing with the hashtable *products* that it defines as a private static variable. Therefore it should provide methods to (key for the letters between brackets: A = Administrator, UU = Unauthenticated User, AU = Authenticated User):

- add products (A)
- edit products (A)
- delete products (A)
- list products (A, UU, AU)
- search products (A, UU, AU)
- view product details (UU, AU)

Searching for products should be possible on all the fields when the administrator is doing a search, and on all but the “dangerous” fields (like the real path of the document, the real file name and the virtual base directory) when a customer is executing a search.
When an (unauthenticated) customer asks to see product details, those “dangerous” fields should not be shown either.

7.3.5 Users servlet class

The products servlet class should do all interfacing with the hashtable *customers* that it defines as a private static variable. Therefore it should provide methods to (key for the letters between brackets: A = Administrator, UU = Unauthenticated User, AU = Authenticated User):

- add customers (A, UU)
- edit customers (A, AU)
- delete customers (A)
- list customers (A)
- search customers (A)
- login (UU)

The “login” feature deserves some explanation. Because we want returning customers to login using their e-mail address and a password they specified, we have to create some sort of “login” feature that connects the current session with a customer. This way we can allow a user to update his/her personal information and look at the documents/sites he or she purchased rights to, through a call to the *Serve* servlet (see 7.3.7).

7.3.6 ProcessTransactions servlet class

The *ProcessTransactions* servlet class should do all interfacing with the hashtable *transactions* that it defines as a private static variable. Therefore it should provide methods to (key for the letters between brackets: A = Administrator, UU = Unauthenticated User, AU = Authenticated User):

- add transactions (A, AU)
- edit transactions (A)
- delete transactions (A)
- list transactions (A)
- search transactions (A)
- process transactions (A)

A transaction is added by an authenticated user when a purchase is made. To keep the transaction database as simple as possible, a separate transaction is created for each purchased product. So if a customer buys rights to 2 documents in one purchase, this will show up as 2 transactions in the database.
The listing of transactions should be possible either including or excluding processed transactions.
The processing of transactions should be no more than a special type of listing the transactions: it should be a list of transactions whose *processed* field has not been set. This method should display the first unprocessed transaction, allowing to change the number of copies bought and the amount to be charged, and allowing the processed flag to be set. A "next" button should allow access to the next unprocessed transaction, until there are no more left.

7.3.7 Serve servlet class

When a customer purchases rights to a document/site, he must be able to access it. This is done through the *Serve* class. This class *serves* the documents/sites to the customers. It must have a *list* method which lists all the documents/sites the customer has bought rights to, and a *file* method that serves the actual files to the customer - provided he or she has the access rights.

This sounds simpler than it is, because for html files, it cannot simply read the file from disk and pass it to the web browser. No, because we want to support the serving of complete sites to customers, it must parse all html files to replace all relative links with a link to the *file* method followed by the original relative link.

This means that when a relative link to another page in the site is encountered, the Serve class should make sure that that link is changed to something it will understand when this link is activated. For instance, when a link to an image in a *graph* subdirectory is made, the HTML source looks like this: *<IMG src="graph/image.gif">.* If the url of the referring

document is *http://some.server.name/index.html*, then the browser will request the file *http://some.server.name/graph/image.gif* when it encounters the IMG tag. In our case, the referring document's URL is something like
http://some.server.name/servlets/Serve/file?basedir/index.html, so if the IMG tag would be unchanged by the *Serve* class, the browser would request the image with the URL *http://some.server.name/servlets/Serve/graph/image.gif*. This would result in an error; what we want is a link to *http://some.server.name/servlets/Serve/file?basedir/graph/image.gif.* Consequently, the *Serve* class must parse the html files it serves, and - in this case - change the IMG tag to <*IMG src="/servlets/Serve/file?basedir/graph/image.gif"*>. So we must add a prefix to the relative links.

Things get more complicated when the fact is taken into account that the use of quotes to delimit the relative url is not compulsory. Also, when a relative url looks like in this tag <*A href="#anchor"*>, i.e. a link to an anchor in the current html file, it should be replaced like this <*A href=""/servlets/Serve/file?basedir/index.html#anchor"*>, assuming that we are still in the *index.html* document. So in this case, the filename of the document should be prefixed as well. Another point is that absolute links (http://some.server.name/...) should not affected, and therefore these should only be used to link to other servers. Good HTML writing requires all local links to be relative anyway, so when a site is written properly, there should not be any problem. Finally, we have to consider that there are several parameters of tags that need parsing; there are the *src* parameter (for images, links to scripts and frames), the *href* parameter (the classical links, the use of external Cascading Style Sheets (CSS) files,...), the *codebase* parameter (java applets) and the *background* parameter (the *body* tag). Also, because HTML is still evolving, the parsing algorithm should be written in such a way that it is easy to add parameters to parse.
Therefore, the parsing of html files should be done in a separate class, *ServeParser*, explained below.

For the *Serve* class, each product has three important fields: a filename as it is stored on disk, a realBaseDir and a virtualBaseDir. More information on these three fields can be found in section 8.2.1, Product.java.
Each product has a filename, and purchasing rights to the product should give rights to all subdirectories and files in the base directory of the product. This allows complete sites of multiple documents to be served by the *Serve* servlet, but it also introduces a potential security risk: if the administrator would change the *realBaseDir* of a product to the root of a drive, all the files on that drive are accessible to anyone who purchases rights to that product. It is read-only access of course, and only available if you know the exact filename and path of the file you want to see, but even then this is not desired. Therefore the administrator should be very careful when designing a directory structure for the e-commerce site. I suggest putting all documents/sites under the same directory, e.g. /doc/ under the root. Then each document and each site should be given a separate subdirectory under that /doc/

directory, so that there can be no access to files that should not be accessible (i.e. system files or other documents/sites).
Trying to get access to files above the *realBaseDir* of a product using a *../../filename* construction must not be possible; there should be a protection against this in the *file* method.
To make sure that documents/sites that are served through this application are not available for free through another web server, the root directory for all documents/sites should not be under the document root directory of any other web server on the machine!

The *Serve* servlet class should also make sure that no more simultaneous views of documents/sites by the same user are allowed than the number of copies that he or she purchased.

7.3.8 ServeParser class

This class should parse files. It should be the responsibility of the *Serve* class to feed only html files to the *ServeParser* class. It should react on the following strings (both in upper- and lowercase):

- HREF=
- SRC=
- CODEBASE=
- BACKGROUND=

But not on (also in upper- and lowercase):

- HREF=HTTP:// or HREF="HTTP:// or HREF=FTP:// or HREF="FTP:// or HREF=MAILTO: or HREF="MAILTO or HREF=NEWS: or HREF="NEWS:
- SRC=HTTP:// or SRC="HTTP:// or SRC=FTP:// or SRC="FTP://
- CODEBASE=HTTP:// or CODEBASE="HTTP://
- BACKGROUND=HTTP:// or BACKGROUND="HTTP://
- all these strings, but with HTTPS instead of HTTP

When one of the above four strings is found in a file, the relative links should be replaced by a relative link to the *file* method in the *Serve* servlet, followed by the original link. This way, all access to the site - the parsing will not change anything when there is only one document because there won't be any relative links in that document - will go through the *file* method of the *Serve* servlet.
Because HTML is still evolving, this class should be written in such a way that it is easy to add extra strings to react on.

7.3.9 Management servlet class

This class should only be a "front-end" for the administration methods that exist in the other servlets (Products, Users and ProcessTransactions). It should not be possible to call these methods directly from the web through these other servlets, because authentication should be done first by the *Management* servlet. This servlet only provides links to the administration methods in the other servlets, after the authentication has been done. The management servlet should only be called over a secure connection to ensure that the login

information and the sensitive information in the databases (e.g. credit card numbers) stays safe.

7.3.10 Cart servlet class

The *Cart* servlet class should do all interfacing with the *shopping cart* that is stored in the session. Therefore it should provide methods to (key for the letters between brackets: A = Administrator, UU = Unauthenticated User, AU = Authenticated User):

- add products to the cart (UU, AU)
- edit the cart contents (UU, AU)
- view the cart contents (UU, AU)
- delete items from/the entire cart (UU, AU)
- display a “are you sure” screen before saving transaction (AU)

The shopping cart is stored in the current session. Because the customers should be able to look around as much as possible before they have to supply their personal information, the session must not necessarily be connected to a customer to fill the shopping cart. When it is not, pressing the “buy now” button should first ask the customer to log in, and then display the “are you sure” screen. If the customer is sure, the transaction(s) will be added to the transaction database via the methods provided in the *ProcessTransaction* servlet.

8. Implementation

8.1 Preamble

All software on the server is written in Java. The E-commerce solution is made up of 6 servlets, 3 data classes, 1 filter class, 1 sorter class and 1 credit card validation class. The data classes are called Product.java, Customer.java and Transaction.java, and they hold respectively an object for a product, an object for a customer and an object for a transaction.
The 6 servlets are Products.java, Users.java, ProcessTransactions.java, Serve.java, Management.java and Cart.java.
The filter class is ServeParser.java, the sorter class is Sorter.java, and the credit card validation class is CheckCC.java.
There is also one JavaScript file, CustomerDetailsFormCheck.js, that holds the functions used for form validation on the client.

8.2 The source code explained

8.2.1 Product.java

Product.java is a simple class that defines an object for a product, and provides 2 methods for initialisation of a new product object. The defined fields for the object Product are: *id, name, type, manufacturer, price, currency, dataAvailable, dateExpired, description, keywords, fileName, virtualBaseDir* and *realBaseDir. A*ll these fields, except the *id* field, are stored as strings.

id

The id of the product. Used by the servlets to identify the products. This field is never editable by the customer or manager, it is always defined via the *Products* (see 8.2.4) Servlet.

name

The name of the product. Used by users and manager to identify products.

type

The type of the product. This field can later be used to determine how to handle the product; right now it is only displayed on the site, and one can search for products of a certain type.

manufacturer

The manufacturer of the product. In the case of products or sites, this field would better have been called "author", but this name might be more correct when the site gets expanded to sell non-document type products.

price

The price of the product.

currency

The currency of the product. This field has been provided to allow easy support for multiple currencies in the future; right now it is not used except for displaying on forms. Since all calculations are done in Pound Sterling, it is advisable always to fill in £ in this field, to avoid confusion.

dateAvailable

The date this product will be available. This field has been provided to allow easy support for date-limited products (e.g. online courses), but this feature is not implemented as of now.

dateExpired

The date this product will be expired. This field has been provided to allow easy support for date-limited products (e.g. online courses) , but this feature is not implemented as of now.

description

The description of this product. This can be a multiple line description.

keywords

A string of comma separated keywords. Handy for use in searches.

fileName

The filename (without path!) of the file containing the product on the server (assuming the product is a document or site).

realBaseDir

The real path on the server of the file containing the product (e.g. /docs/). It is advisable always to use forward slashes, even on a Windows system. The *realBaseDir* may end in a slash or not, both are allowed, but it should always begin with a forward slash. A hardcoded driveletter and base directory are always prefixed when this field is used; they are specified in the *Products* servlet (see section 8.2.4).

virtualBaseDir

The virtual path on the server of the file containing the product (e.g. javaTutorial/). This path is used to request the document or site for viewing. The customer will request a URL looking like: http://some.server.name/Servlet/Serve?vbd=javaTutorial/index.html. In this URL, “Serve” is the servlet that serves the documents and sites to the customer. “Servlet” is the virtual base directory for your servlets, as specified in the server software (e.g. NES). “vbd” is short for Virtual Base Directory, and is the parameter supplied to the Serve servlet to specify which document or site is requested. More information can be found in the description of the Serve servlet. The *virtualBaseDir* may end in a slash or not, both are allowed.

The defined methods for the object Product are: int *getNextID(),* and *void setNextID(int id)*

int getNextID()

Returns the next available Product ID. This method is called when a new Product object is created.

void setNextID(int id)

Set the next available Product ID. This method is called from the readHashFile() method in the Products servlet (see 8.2.4), to set the next available ID after the hashtable with products has been read from file.

The code of the Product.java class can be found on the disk accompanying this report.

8.2.2 Customer.java

Customer.java is a simple class that defines an object for a customer, and provides 2 methods for initialisation of a new customer object. The defined fields for the object Customer are: *id, firstName, lastName, emailAddress, password, telephoneNumber, creditCardName, creditCardType, creditCardNumber, creditCardExpiryMonth, creditCardExpiryYear, creditCardZIP, addressLine1, addressLine2, addressCity, addressZIP, addressProvince, addressCountry* and *purchasedRights.* All these fields except the *id, creditCardExpiryMonth* and *creditCardExpiryYear* fields are stored as strings!

id

The id of the customer. Used by the servlets to identify the customers. This field is never editable by the customer or manager, it is always defined via the *Users* (see 8.2.5) servlet.

firstName

The first name of the customer. This field is required when the user fills out the form; there are no restrictions when the administrator adds or edits a customer-record.

lastName

The last name of the customer. This field is required when the user fills out the form; there are no restrictions when the administrator adds or edits a customer-record.

emailAddress

The e-mail address of the customer. Once registered (i.e. after the first purchase), the customer can access the documents/sites he or she purchased rights to by logging in with the *emailAddress* as login name and the *password* (see below) as password. The customer should also login using these credentials when doing a subsequent purchase, to avoid having to specify all the personal information again. This field is required when the user fills out the form; there are no restrictions when the administrator adds or edits a customer-record.

telephoneNumber

The telephone number of the customer. Not required.

creditCardName

The name of the credit card holder. This name must not necessarily be the same as the combined string *firstName + lastName,* but that value is where it defaults to, when a new user fills out the form. This field is required when the user fills out the form; there are no restrictions when the administrator adds or edits a customer-record.

creditCardType

The type of credit card the customer has. The options include: VISA, MASTERCARD, DISCOVER, AMERICAN EXPRESS, JCB, DINERS and ENROUTE. These types of creditcards are supported because validation code for them was available on the Internet. This field is required when the user fills out the form; there are no restrictions when the administrator adds or edits a customer-record.

creditCardNumber

The credit card number of the customer. This field is required when the user fills out the form, and there is also a validity check. This check is also done when the administrator adds or edits a customer-record.

creditCardExpiryMonth

The month the credit card will expire. This field is required when the user fills out the form; there are no restrictions when the administrator adds or edits a customer-record. When a new Customer object is created, creditCardExpiryMonth is set to 0. Possible values range from 0 (January) to 11 (December).

creditCardExpiryYear

The year the credit card will expire. This field is required when the user fills out the form; there are no restrictions when the administrator adds or edits a customer-record. When a new Customer object is created, creditCardExpiryYear is set to 0. The creditCardExpiryYear is saved as a 4 digit number to avoid Year 2000 (Y2K) problems.

creditCardZIP

The postal code of the credit card.

addressLine1

The first address line of the customer.

addressLine2

The second address line of the customer.

addressCity

The city of the customer.

addressZIP

The postal code of the customer.

addressProvince

The province of the customer.

addressCountry

The country of the customer.

purchasedRights

The rights the customer has purchased. This field is obviously never directly editable by the user, but the manager can edit it. The format of this field is: id**=**quantity**&**id**=**quantity**&**... where id is the product id and quantity the number of purchased copies. The quantity field determines the amount of simultaneous logins that this customer is allowed for the product specified by the id field.

The defined methods for the object Customer are: int *getNextID(),* and *void setNextID(int id)*

addressCountry

Returns the next available Customer ID. This method is called when a new Customer object is created.

void setNextID(int id)

Set the next available Customer ID. This method is called from the readHashFile() method in the Users servlet (see 8.2.5), to set the next available ID after the hashtable with customers has been read from file.

The code of the Customer.java class can be found on the disk accompanying this report.

8.2.3 Transaction.java

Transaction.java is a simple class that defines an object for a transaction, and provides 2 methods for initialisation of a new transaction object. The defined fields for the object Transaction are: *id, date, userId, productId, quantity, amount, currency* and *processed*. All fields are ints, except for the Date *date*, the String *currency* and the boolean *processed*.

id

The id of the transaction. Used by the servlets to identify the transactions. This field is never editable by the customer or manager, it is always defined via the *ProcessTransactions* (see 8.2.6) Servlet.

date

The date and time of the transaction. This field is stored as a Date object.

userId

The id of the customer that is charged for the transaction.

productId

The id of the product that is purchased in the transaction. When a customer purchases several products at the same time, a separate transaction is created for every product.

quantity

The amount of copies the customer buys.

amount

The amount that will be charged to the credit card.

currency

The currency of the amount. This field will always be pound Sterling; it is provided for future implementation of a multiple currency system.

processed

This boolean field specifies if this transaction has been processed. Set to false when the transaction is created, it should be set to true when the administrator processes the transaction. This field is used in the *ProcessTransaction* servlet to make a distinction between unprocessed and processed transactions.

The defined methods for the object *Transaction* are: int *getNextID(),* and *void setNextID(int id)*

int getNextID()

Returns the next available Transaction ID. This method is called when a new Transaction object is created.

void setNextID(int id)

Set the next available Transaction ID. This method is called from the readHashFile() method in the Users servlet (see 8.2.5), to set the next available ID after the hashtable with transactions has been read from file.

The code of the Transaction.java class can be found on the disk accompanying this report.

8.2.4 Products.java

Products.java is the servlet class that does all the interfacing with the hashtable that contains the products. Other servlets can only access the - private - hashtable *products* through methods of the *Products* class.

8.2.4.1 Variables

The *Products* class defines the following private variables:

- *boolean debug = false;*
- *static Hashtable products = new Hashtable();*
- *static String filename;*

Two of these variables are *static*. This means that they are not variables of an instance of the class, but that they are variables of the class itself and consequently have the same value for all the instances.

The boolean *debug* is not static. It is a variable that determines if debug-output should be written to the log files or not. It is set in the *init()* method of the servlet, depending on how it was called (as http://some.server.name/Servlet/Products/some/command -> debug false, or as http://some.server.name/Servlet/ProductsDebug/some/command -> debug true). For more information about the calling of servlets from a URL, refer to section 5.2.
The *products* hashtable holds all information about the products in memory; it is a hashtable with *Product* objects (see 8.2.1).
The String *filename* holds the filename of the database with products, as specified in the *servlets.properties* file. It is set in the *init()* method. For more information about the *servlets.properties* file refer to section 5.2.

8.2.4.2 Methods

The *Products* class provides the following public methods:

- *void init(ServletConfig config)*
- *void destroy()*
- *void listProducts(ServletOutputStream out, String path, boolean admin)*
- *static String getServletDir(String path)*
- *void editProduct(ServletOutputStream out, HttpServletRequest req)*
- *static Product searchById(int searchId)*
- *void printProductList(ServletOutputStream out, int productId)*
- *static Product searchByVirtualBaseDir(String virtualBaseDir)*
- *void doSearch(ServletOutputStream out, HttpServletRequest req, boolean admin)*
- *void processChange(ServletOutputStream out, HttpServletRequest req)*
- *void doPost (HttpServletRequest req, HttpServletResponse res)*
- *void doGet (HttpServletRequest req, HttpServletResponse res)*
- *ServletOutputStream startOutput(HttpServletRequest req, HttpServletResponse res)*

It also provides the following private methods:

- *void saveHashFile(String filename)*
- *int findHighestId()*
- *void readHashFile(String filename)*
- *void addToHash(Product p)*
- *void deleteFromHash(Product p)*
- *void writeSearchButton(ServletOutputStream out, String path, String action)*
- *void viewProduct(ServletOutputStream out, HttpServletRequest req)*
- *void writeBeginningOfForm(ServletOutputStream out, String path, String formAction, boolean admin)*
- *void writeEndOfForm(ServletOutputStream out)*
- *void writeForm(ServletOutputStream out, Product p, String path, String formAction, boolean admin)*
- *Product searchProduct(HttpServletRequest req)*
- *static String extractVirtualBaseDir(String vbdParam)*
- *void processRequest(HttpServletRequest req, HttpServletResponse res)*

- *void endOutput(ServletOutputStream out)*

These methods are discussed in detail below.

8.2.4.2.1 public void init(ServletConfig config)

Called by the web server when the servlet is just loaded. In this method the class variables "debug" and "filename" are set, with values as specified in the *servlet.properties* file (see section 5.2). Then the *products hashtable* is read from file.
There is one parameter, *config*, of the type *ServletConfig.* It contains the configuration information and is passed to the *init* method by the server.

8.2.4.2.2 public void destroy(ServletConfig config)

Called by the web server when the server wants to drop the servlet from the Java Virtual Machine (JVM). Clears the hashtable in memory.
There is one parameter, *config*, of the type *ServletConfig.* This parameter contains the configuration information and is passed to the *destroy* method by the server.

8.2.4.2.3 private void saveHashFile(String filename)

Called when the Hashtable has to be written to file. This method saves the *products* hashtable with one command, thanks to serialization!
There is one parameter, *filename*, of the type *String,* containing the filename of the *products* file on disk.

8.2.4.2.4 private int findHighestId()

This method returns the highest Id in use in the *products* hashtable + 1. It is called from readHashFile(), because when readHashFile() is called, on load of the *products* database file from disk, we have to initialise the next available id for the Product class. Therefore, we have to find the highest Id in use in the hashtable, increase it with 1 and return it.

8.2.4.2.5 private void readHashFile(String filename)

Called when the Hashtable has to be read from file. Reads the complete hashtable with one command, thanks to serialization, and then sets the next available Product Id in the Product class by using the *findHighestId()* method.
There is one parameter, *filename*, of the type *String.* It contains the filename of the *products* file on disk.

8.2.4.2.6 private void addToHash(Product p)

This method adds a new product to, or updates a product in the *products* hashtable.

There is one parameter, *p*, of the type *Product*, containing the *Product* object to be added or updated. The *id* field of the *Product* object is used as the key in the hashtable.

8.2.4.2.7 private void deleteFromHash(Product p)

This method deletes a product from the *products* hashtable.
There is one parameter, *p*, of the type *Product.* This parameter contains the *Product* object to be deleted. The *id* field of the *Product* object is used as the key in the hashtable.

8.2.4.2.8 private void writeSearchButton(ServletOutputStream out, String path, String action)

This method writes a html form with a search button and a set of 2 radiobuttons to *ServletOutputStream out*. The radiobuttons specify the type of search: AND or OR.
This method uses a parameter *action*, which can be either the *String* "Search" or the *String* "SearchNow". The *action* parameter equals "Search" when the search form must be displayed, and "SearchNow" when the form search results must be displayed.
There are three parameters, the first one is *out*, of the type *ServletOutputStream.* It is the stream the output of the method is written to. The second parameter is *path*, a *String* that contains the partial URL of the servlet (without the server name and without query-string). It is used to specify the form action. The third parameter is *action*, another *String*, whose function is explained higher.

8.2.4.2.9 public void listProducts(ServletOutputStream out, String path, boolean admin)

This method displays a list of available products, sorted alphabetically in the form “*p.name* by *p.manufacturer*” . For an explanation of these fields, see section 8.2.1. Depending on the *admin* parameter, a *boolean* that can be either true or false, it is decided which type of list is created. When the administrator makes a request for a list of products, always through the *Management* servlet (see 8.2.9), the *admin* parameter is true. In that case, a button is displayed that allows the creation of new *products*, and the links in the product list point to *editProduct*, in stead of *viewProduct* when a normal customer requests a list.
There are three parameters, the first one is *out*, of the type *ServletOutputStream.* It is the stream the output of the method is written to. The second parameter is *path*, a *String* that contains the partial URL of the servlet (without the server name and without query-string). It is used to specify the form action. The third parameter is *admin*, a *boolean*, whose function is explained higher.

8.2.4.2.10 public static String getServletDir(String path)

This method extracts the virtual directory of the servlet out of the URL. For example, when the products servlet would be called as *http://some.server.name/servlet/Products*, this method would return *servlet/ .*

There is one parameter, *path*, of the type *String*, containing the partial URL of the servlet (without the server name and without query-string).

8.2.4.2.11 public void viewProduct(ServletOutputStream out, HttpServletRequest req)

This method is called when a user wants to view a particular product. It writes out a html page with the product name, type, manufacturer, price, date available, date expired, keywords and description. Links on the page allow to go to the main shop page, to the product search page, to add this product to the shopping cart, and to view the contents of the current shopping cart.
There are two parameters, the first one is *out*, of the type *ServletOutputStream*, holding the stream the output of the method is written to. The second parameter is *req*, a *HttpServletRequest* that contains the request information, like the parameters passed to the servlet, ...

8.2.4.2.12 private void writeBeginningOfForm(ServletOutputStream out, String path, String formAction, boolean admin)

This method writes the a form header with a form action as specified in the method parameters *formAction* and *path* and also depending on the *admin* parameter. It also writes the <TABLE> tag.
There are four parameters, the first one is *out*, of the type *ServletOutputStream*. It is the stream the output of the method is written to. The second parameter is *path*, a *String* that contains the partial URL of the servlet (without the server name and without query-string). The third parameter is *formAction*, a *String* that is used to specify the form action. The fourth parameter is *admin*, a *boolean*, true if this request is made by the administrator of the site, and false otherwise.

8.2.4.2.13 private void writeEndOfForm(ServletOutputStream out)

This method writes the </FORM> tag to *ServletOutputStream out.*
There is one parameter, *out*, of the type *ServletOutputStream*, containing the stream the output of the method is written to.

8.2.4.2.14 private void writeForm(ServletOutputStream out, Product p, String path, String formAction, boolean admin)

This method is used to display a search or edit form, depending on the *formAction* parameter. If the *admin* parameter is true, New/Save/Delete buttons are displayed. The administrator can search on more fields (fileName, realBaseDir and virtualBaseDir - for more information see section 8.2.1). This form can only be an editing form for a *Product* object when *admin* is true.
There are five parameters. The first one is *ServletOutputStream out*, the stream the output of the method is written to. The second parameter is *Product p*, the *Product* object to be edited or the *Product* object to be used to store the search information. The third parameter is *path*,

a *String* that contains the partial URL of the servlet (without the server name and without query-string). The fourth parameter is *formAction*, a *String* that is used to specify the form action. The fifth parameter is *admin*, a *boolean*, true if this request is made by the administrator of the site, and false otherwise.

8.2.4.2.15 public void editProduct(ServletOutputStream out, HttpServletRequest req)

This method is called when a product needs to be edited by the administrator. Writes a form with the fields of the product to be edited already filled in using the *writeForm* method, and with save, new, and search buttons at the bottom.
There are two parameters. The first one is *ServletOutputStream out*, the stream the output of the method is written to. The second parameter is *req*, a *HttpServletRequest,* that contains the request information like the parameters passed to the servlet etc.

8.2.4.2.16 private Product searchProduct(HttpServletRequest req)

This method extracts the id parameter from the *HttpServletRequest req*, and tries to find a product with matching id in the products hashtable. Returns the found *Product* object, or returns null if no match is found, or if the id parameter is not specified. This method is called from *editProduct* and *processChange*.
There is one parameter, *req*, a *HttpServletRequest,* that contains the request information like the parameters passed to the servlet etc.

8.2.4.2.17 public static Product searchById(int searchId)

This method finds a product with matching id in the *products* hashtable. It returns the found *Product* object, or returns null if no match is found. Because it is called from the *Cart* Servlet and *Serve* Servlet (see 8.2.10 and 8.2.7), it needs to be public.
There is one parameter, *searchId*, an *int* that contains the id of the product that must be found.

8.2.4.2.18 public void printProductList(ServletOutputStream out, int productId)

This method writes the product list for use in a SELECT form-element to the *ServletOutputStream out.* It is called from the *ProcessTransactions* Servlet (see 8.2.6).
There are two parameters. The first one is *ServletOutputStream out,* the stream the output of the method is written to. The second parameter is *productId*, an *int,* containing the id of the product that should be preselected in the SELECT form-element, or -1 if none should be pre-selected.

8.2.4.2.19 private static String extractVirtualBaseDir(String vbdParam)

This method extracts the virtualBaseDir from the vbdParam; i.e. the part before the first forward slash. After that slash, a directory/file structure can be specified. The *vbdParam* parameter looks for example like: "WardSite/index.html", and is extracted in the handleFileRequest method in the *Serve* servlet (see 8.2.7) from "http://some.server.name/Servlet/Serve/file?vbd=WardSite/index.html". This method is called from the static method *searchByVirtualBaseDir*, that is why it must be static.
There is one parameter, *vbdParam*, a *String,* containing the information to be parsed.

8.2.4.2.20 public static Product searchByVirtualBaseDir(String virtualBaseDir)

This method is used to find a product with matching *virtualBaseDir* in the *products* hashtable. It returns the found *Product* object, or null if no match is found. It is called from *Serve* Servlet (see 8.2.7).
There is one parameter, *virtualBaseDir*, a *String,* containing the virtualBaseDir to be searched for.

8.2.4.2.21 public void doSearch(ServletOutputStream out, HttpServletRequest req, boolean admin)

If a search request occurs, this method processes the search. This means either display the search form, or do the search and display the results, depending on the action parameter of the pressed button (passed as a form variable via the *req HttpServletRequest*).
There are three parameters. The first one is *ServletOutputStream out,* the stream the output of the method is written to. The second parameter is *req*, a *HttpServletRequest,* containing the request information like the parameters passed to the servlet etc.The third parameter is

admin, a *boolean*, depending on which more or less fields are showed. See section 8.2.4.2.14, *writeForm*, for more information.

8.2.4.2.22 public void processChange(ServletOutputStream out, HttpServletRequest req)

If a change request occurs, this method processes the save (new or update) or delete request. It can obviously only be called by the administrator.
There are two parameters. The first one is *ServletOutputStream out,* the stream the output of the method is written to. The second parameter is *req*, a *HttpServletRequest,* containing the request information like the parameters passed to the servlet etc.

8.2.4.2.23 public void doPost (HttpServletRequest req, HttpServletResponse res

This method is called by the web server when a HTTP POST request is made. It checks if the length of the *HttpServletRequest req* is smaller than 8192 bytes; if it is, the *processRequest* method (see 8.2.4.2.25) is called. If not, the request is denied. This is to limit a typical “denial of service attack”, to which some web servers are still vulnerable. The concept is to flood the webserver with a very long *HttpServletRequest*, thereby making it crash.
There are two parameters, of which the first one is *req*, a *HttpServletRequest* containing the request information like the parameters passed to the servlet etc. The second parameter is *res,* a *HttpServletResponse* object used to specify the HTTP header information etc.

8.2.4.2.24 public void doGet (HttpServletRequest req, HttpServletResponse res

This method is called by the web server when a HTTP GET request is made. It simply calls the *processRequest* method (see 8.2.4.2.25). There is no risk for the “denial of service” attack described in 8.2.4.2.23 when GET is used to retrieve information from the server.
There are two parameters, of which the first one is *req*, a *HttpServletRequest* containing the request information like the parameters passed to the servlet etc. The second parameter is *res,* a *HttpServletResponse* object used to specify the HTTP header information etc.

8.2.4.2.25 private void processRequest(HttpServletRequest req, HttpServletResponse res)

This method takes a look at the path information and parameters and performs the requested action. This is the core method of this servlet, as it decides which of the above methods should be called when. A lot of the above methods are not called from here though, but from other servlets (notably the *Management* Servlet, see 8.2.9).
There are two parameters. The first one is *req*, a *HttpServletRequest,* containing the request information like the parameters passed to the servlet etc. The second parameter is *res,* a *HttpServletResponse* object used to specify the HTTP header information etc.

8.2.4.2.25 public ServletOutputStream startOutput(HttpServletRequest req, HttpServletResponse res)

This method sets the HTTP content type and writes out the HTML header. It returns a *ServletOutputStream* object where the output of the servlet should be written to.
There are two parameters. The first one is *req*, a *HttpServletRequest,* containing the request information like the parameters passed to the servlet etc. The second parameter is *res,* a *HttpServletResponse* object used to specify the HTTP header information etc.

8.2.4.2.26 private void endOutput(ServletOutputStream out)

This method writes the end of the HTML document: it writes the </BODY> and </HTML> tags.
There is one parameter, *out*, of the type *ServletOutputStream*, which is the stream the output of the method is written to.

The code of the *Products* Servlet class can be found on the disk accompanying this report.

8.2.5 Users.java

Users.java is the servlet class that does all the interfacing with the hashtable that contains the customers. Other servlets can only access the - private - hashtable *customers* through methods of the *Users* class.

8.2.5.1 Variables

The *Users class* defines the following variables/constants:

- *private boolean debug = false;*
- *private static Hashtable customers = new Hashtable();*
- *public static String filename;*
- *public final String redirectToStr = "redirectTo";*
- *public final String userIdStr = "userId";*

For information about the *debug* variable, see 8.2.4.1.
The *customers* hashtable holds all information about the products in memory; it is a hashtable with *Customer* objects (see 8.2.2).
The String *filename* holds the filename of the database with customers, as specified in the *servlets.properties* file. It is set in the *init()* method. For more information about the *servlets.properties* file refer to section 5.2.
The String *redirectToStr* is a final, this is a constant. Therefore there is no need to declare it static - it has a fixed value for all the instances of the class anyway. This constant is merely a placeholder; it defines a string that is used as a key to save and retrieve redirection information in a session.
The String *userIdStr* is also a final. This constant is merely a placeholder; it defines a string that is used as a key to save and retrieve the user id of a customer in a session.

8.2.5.2 Methods

The *Users* class provides the following public methods:

- *void init(ServletConfig config)*
- *void destroy()*
- *void saveHashFile(String filename)*
- *void addToHash(Customer c)*
- *void listUsers(ServletOutputStream out, HttpServletRequest req)*
- *void editUser(ServletOutputStream out, HttpServletRequest req)*
- *void processChange(ServletOutputStream out, HttpServletRequest req, HttpServletResponse res)*
- *static Customer searchByUserId(Integer userId)*
- *void printProductList(ServletOutputStream out, int userId)*
- *Customer searchByEmailAddress(String emailAddress)*
- *void writeLoginForm(ServletOutputStream out, HttpServletRequest req)*
- *void doSearch(ServletOutputStream out, HttpServletRequest req)*
- *void doPost (HttpServletRequest req, HttpServletResponse res)*
- *void doGet (HttpServletRequest req, HttpServletResponse res)*
- *ServletOutputStream startOutput(HttpServletRequest req, HttpServletResponse res)*

It also provides the following private methods:

- *int findHighestId()*
- *void readHashFile(String filename)*
- *void deleteFromHash(Customer c)*
- *Customer checkUser(HttpServletRequest req, HttpServletResponse res, ServletOutputStream out)*
- *void handleExisting(ServletOutputStream out, String formAction, String path)*
- *void showHelpScreen(ServletOutputStream out, String formAction, String path)*
- *void sendEmail(HttpServletRequest req, ServletOutputStream out)*
- *void writeBeginningOfForm(ServletOutputStream out, String path, String formAction)*
- *void writeEndOfForm(ServletOutputStream out)*
- *void writeForm(ServletOutputStream out, Customer u, String formAction, String path, boolean admin)*
- *Customer processNewCustomerInfo(HttpServletRequest req, StringBuffer messageBuff)*
- *void writeSearchButton(ServletOutputStream out, String path, String action)*
- *void writeSearchForm(ServletOutputStream out, String path)*
- *void logoutUser(HttpServletRequest req, ServletOutputStream out, HttpSession mySession)*
- *void processRequest(HttpServletRequest req, HttpServletResponse res)*
- *void endOutput(ServletOutputStream out)*

These methods are discussed in detail below.

8.2.5.2.1 public void init(ServletConfig config)

Called by the web server when the servlet is just loaded. In this method the class variables "debug" and "filename" are set, with values as specified in the *servlet.properties* file (see section 5.2). Then the *customers* hashtable is read from file.
There is one parameter, *config*, of the type *ServletConfig*, containing the configuration information. It is passed to the *init* method by the server.

8.2.5.2.2 public void destroy(ServletConfig config)

Called by the web server when the server wants to drop the servlet from the JVM. Clears the hashtable in memory.
There is one parameter, *config*, of the type *ServletConfig*, containing the configuration information. It is passed to the *destroy* method by the server.

8.2.5.2.3 public void saveHashFile(String filename)

Called when the Hashtable has to be written to file. This method saves the *customers* hashtable with one command, thanks to serialization! This method must be public because it is called from the *ProcessedTransactions* Servlet to save the *purchasedRights* field (see 8.2.2).
There is one parameter, *filename*, of the type *String*, which contains the filename of the *products* file on disk.

8.2.5.2.4 private int findHighestId()

This method returns the highest Id in use in the hashtable *customers* + 1. It is called from *readHashFile()*, because when *readHashFile()* is called, on load of the *customers* database file from disk, we have to initialise the next available id for the *Customer* class. Therefore, we have to find the highest Id in use in the hashtable, increase it with 1 and return it.

8.2.5.2.5 private void readHashFile(String filename)

Called when the *customers* hashtable has to be read from file. Reads the complete hashtable with one command, thanks to serialization, and then sets the next available *Customer* Id in the *Customer* class by using the *findHighestId()* method.
There is one parameter, *filename*, of the type *String*, which contains the filename of the *customers* file on disk.

8.2.5.2.6 public void addToHash(Customer c)

This method adds a new customer to, or updates a customer in the *customers* hashtable.
There is one parameter, *c*, of the type *Customer*, which contains the *Customer* object to be added or updated. The *id* field of the *Customer* object is used as the key in the hashtable.

This method must be public because it is called from *ProcessedTransactions* Servlet to save the *purchasedRights* field (see 8.2.2).

8.2.5.2.7 private void deleteFromHash(Customer c)

This method deletes a customer from the *customers* hashtable.
There is one parameter, *c*, of the type *Customer*, which contains the *Customer* object to be deleted. The *id* field of the *Customer* object is used as the key in the hashtable.

8.2.5.2.8 void listUsers(ServletOutputStream out, HttpServletRequest req)

This method displays the list of customers, sorted alphabetically in the form "c*.lastName,* c*.firstname*" . For an explanation of these fields, see section 8.2.2. This method can only be called by the administrator, through the *Management* Servlet (see 8.2.9). A button is displayed that allows the creation of new *products*, and the links in the product list point to *editUser*.
There are two parameters. The first one is *ServletOutputStream out*, the stream the output of the method is written to. The second parameter is *path*, a *String* that contains the partial URL of the servlet (without the server name and without query-string). It is used to specify the form action.

8.2.5.2.9 public void editUser(ServletOutputStream out, HttpServletRequest req)

This method is called when a customer needs to be edited by the administrator. Writes a form with the fields of the customer to be edited already filled in using the *writeForm* method, and with save, new, and search buttons at the bottom.
There are two parameters. The first one is *ServletOutputStream out*, the stream the output of the method is written to. The second parameter is *req*, a *HttpServletRequest* that contains the request information like the parameters passed to the servlet etc.

8.2.5.2.10 public void processChange(ServletOutputStream out, HttpServletRequest req, HttpServletResponse res)

If a change request occurs, this method processes the save (new or update) or delete request. It can obviously only be called by the administrator.
There are three parameters. The first one is *ServletOutputStream out*, the stream the output of the method is written to. The second parameter is *req*, a *HttpServletRequest* containing the request information like the parameters passed to the servlet etc. The third parameter, *res*, is a *HttpServletResponse* object used to specify the HTTP header information etc.

8.2.5.2.11 public static Customer searchByUserId(Integer userId)

This method finds a customer with matching id in the *customers* hashtable. It returns the found *Customer* object, or returns null if no match is found. Because it is called from the *Cart* Servlet and *Serve* Servlet (see 8.2.10 and 8.2.7), it needs to be public.
There is one parameter, *userId*, an *Integer,* containing the id of the product that has to be found.

public void viewProduct(ServletOutputStream out, HttpServletRequest req)

This method is called when a user wants to view a particular product. It writes out a html page with the product name, type, manufacturer, price, date available, date expired, keywords and description. Links on the page allow to go to the main shop page, to the product search page, to add this product to the shopping cart, and to view the contents of the current shopping cart.
There are two parameters. The first one is *ServletOutputStream out*, which is the stream the output of the method is written to. The second parameter is *req*, a *HttpServletRequest* that contains the request information like the parameters passed to the servlet etc.

8.2.5.2.12 public void printUserList(ServletOutputStream out, int userId)

This method writes the product list for use in a SELECT form-element to the *ServletOutputStream out.* It is called from the *ProcessTransactions* Servlet (see 8.2.6).
There are two parameters, of which the first one is *ServletOutputStream out*, the stream the output of the method is written to. The second parameter is *userId*, an *int,* containing the id of the customer that should be preselected in the SELECT form-element, or -1 if none should be pre-selected.

8.2.5.2.13 Customer searchByEmailAddress(String emailAddress)

This method is used to find a customer with matching *emailAddress* in the *customers* hashtable. It returns the found *Customer* object, or null if no match is found. It is called from *Serve* Servlet (see 8.2.7).
There is one parameter, *emailAddress*, a *String,* containing the e-mail address to be searched for.

8.2.5.2.14 Customer checkUser(HttpServletRequest req, HttpServletResponse res, ServletOutputStream out)

This method checks the user login. It checks if the supplied password matches with the supplied e-mail address; if so, it returns the corresponding Customer object; otherwise it returns null.
There are three parameters. The first one is *req*, a *HttpServletRequest,* containing the request information like the parameters passed to the servlet etc. The second parameter, *res*, is a *HttpServletResponse* object used to specify the HTTP header information etc. The

third parameter, *out*, of the type *ServletOutputStream*, is the stream where the output of the method is written to.

8.2.4.2.15 public void editProduct(ServletOutputStream out, HttpServletRequest req)

This method is called when a product needs to be edited by the administrator. It writes a form with the fields of the product to be edited already filled in using the *writeForm* method, and with save, new, and search buttons at the bottom.
There are two parameters. The first one is *ServletOutputStream out*, the stream the output of the method is written to. The second parameter is *req*, a *HttpServletRequest* that contains the request information like the parameters passed to the servlet etc.

8.2.5.2.16 void handleExisting(ServletOutputStream out, String formAction, String path)

This method draws a login form for a returning customer, with a field for the e-mail address of the customer, and one for his password. It also provides a link with instructions on what to do when he has forgotten his password.
There are three parameters. The first parameter, *out*, of the type *ServletOutputStream*, is the stream where the output of the method is written to. The second parameter is *formAction*, a *String* that is used to specify the form action. The third parameter is *path*, a *String* that contains the partial URL of the servlet (without the server name and without query-string).

8.2.5.2.17 private void showHelpScreen(ServletOutputStream out, String formAction, String path)

This method explains the user what to do when he forgot his password. On the click of the submit button, the method *sendEmail()* is called.
There are three parameters. The first parameter, *out*, of the type *ServletOutputStream*, is the stream where the output of the method is written to. The second parameter is *formAction*, a *String* that is used to specify the form action. The third parameter is *path*, a *String* that contains the partial URL of the servlet (without the server name and without query-string).

8.2.5.2.18 private void sendEmail(HttpServletRequest req, ServletOutputStream out)

This method sends an e-mail to the user when he has forgotten his password, containing the password. Of course, a check is done first to find out if this e-mail address is registered. In fact, the supplied e-mail address is looked up in the *customers* database, and the matching password is sent to this address. So there is no possible way to steal someone's address using this feature of the software - apart from intercepting the e-mail.
There are two parameters. The first one is *req*, a *HttpServletRequest,* containing the request information like the parameters passed to the servlet etc. The second parameter is *out*, of the type *ServletOutputStream*, which is the stream the output of the method is written to.

8.2.5.2.19 private void writeBeginningOfForm(ServletOutputStream out, String path, String formAction)

This method writes a form header with a form action as specified in the method parameters *formAction* and *path*. It also writes the <TABLE> tag.
There are three parameters. The first one is *out*, of the type *ServletOutputStream*, which is the stream the output of the method is written to. The second parameter is *path*, a *String* that contains the partial URL of the servlet (without the server name and without query-string). The third parameter is *formAction*, a *String* that is used to specify the form action.

8.2.5.2.20 private void writeEndOfForm(ServletOutputStream out)

This method writes the </FORM> tag to *ServletOutputStream out.*
There is one parameter, *out*, of the type *ServletOutputStream*, which is the stream the output of the method is written to.

8.2.5.2.21 private void writeForm(ServletOutputStream out, Customer u, String formAction, String path, boolean admin)

This method is called when customer information has to be recorded. It displays a form that allows filling in new information or editing existing information. Depending on the admin parameter, the output is slightly different: the administrator does not have some of the field restrictions imposed on the users by use of Javascript (no empty required fields,...). The checks for a correct credit card number and a valid expiry year are done even when the administrator is editing user information. The second password field is only displayed when the customer edits his or her information.
There are five parameters. The first one is *ServletOutputStream out*, the stream the output of the method is written to. The second parameter is *c*, of the type *Customer*, containing the *Customer* object to be edited. The third parameter is *formAction*, a *String* that is used to specify the form action. The fourth parameter is *path*, a *String* that contains the partial URL of the servlet (without the server name and without query-string). The fifth parameter is *admin*, a *boolean*, true if this request is made by the administrator of the site, and false otherwise.

8.2.5.2.22 public void writeLoginForm(ServletOutputStream out, HttpServletRequest req)

This method displays a form with two buttons that allow access to the new customer login and the returning customer login.
There are two parameters, of which the first one is *ServletOutputStream out*, the stream the output of the method is written to. The second parameter is *req*, a *HttpServletRequest* that contains the request information like the parameters passed to the servlet etc.

8.2.5.2.23 private Customer processNewCustomerInfo(HttpServletRequest req, StringBuffer messageBuff)

This method is called when customer information should be recorded. It performs a number of checks on the provided information, and returns a list of problems through the *messageBuff* parameter if any are found. Most of these checks are done on the client as well, but we can not trust on that because it is easy enough to avoid these tests! All you have to do is save the form locally, replace the Javascript call by a simple *submit* and there you are. Why still use client tests then? Because they make everything faster, as explained in section 4.6, Programming languages on the client.
If no problems are encountered, this method returns the *Customer* object, otherwise, it returns null.
There are two parameters. The first one is *req*, a *HttpServletRequest,* containing the request information like the parameters passed to the servlet etc. The second parameter is *messageBuff*, a *StringBuffer* that contains all error-messages if problems are found.

8.2.5.2.24 private void writeSearchButton(ServletOutputStream out, String path, String action)

This method writes a html form with a search button and a set of 2 radiobuttons to *ServletOutputStream out*. The radiobuttons specify the type of search: AND or OR.
This method uses a parameter *action*, which can be either the *String* "Search" or the *String* "SearchNow". The *action* parameter equals "Search" when the search form must be displayed, and "SearchNow" when the form search results must be displayed.
There are three parameters. The first one is *out*, of the type *ServletOutputStream*, which is the stream the output of the method is written to. The second parameter is *path*, a *String* that contains the partial URL of the servlet (without the server name and without query-string). It is used to specify the form action. The third parameter is *action*, another *String*, whose function is explained higher.

8.2.5.2.25 private void writeSearchForm(ServletOutputStream out, String path)

This method displays the customers search form. The fields that can be searched on are firstName, lastName and creditCardNumber.
There are three parameters. The first one is *out*, of the type *ServletOutputStream*, which is the stream the output of the method is written to. The second parameter is *path*, a *String* that contains the partial URL of the servlet (without the server name and without query-string). It is used to specify the form action.

8.2.5.2.26 public void doSearch(ServletOutputStream out, HttpServletRequest req)

If a search request occurs, this method processes the search. This means either display the search form, or do the search and display the results, depending on the action parameter of the pressed button (passed as a form variable via the *req HttpServletRequest*).

There are two parameters, of which the first one is *out*, of the type *ServletOutputStream.* It is the stream the output of the method is written to. The second parameter is *req*, a *HttpServletRequest,* containing the request information like the parameters passed to the servlet etc.

8.2.5.2.27 private void logoutUser(HttpServletRequest req, ServletOutputStream out, HttpSession mySession)

This method logs out the customer by invalidating the current session.
There are three parameters, of which the first one is *out*, of the type *ServletOutputStream.* It is the stream the output of the method is written to. The second parameter is *req*, a *HttpServletRequest,* containing the request information like the parameters passed to the servlet etc. The third parameter is *mySession*, of the type *HttpSession*. It holds the current Session object, or null if there is none.

8.2.5.2.28 public void doPost (HttpServletRequest req, HttpServletResponse res)

This method is called by the web server when a HTTP POST request is made. It checks if the length of the *HttpServletRequest req* is smaller than 8192 bytes; if it is, the *processRequest* method (see 8.2.5.2.30) is called. If not, the request is denied. This is to limit a typical “denial of service attack”, to which some web servers are still vulnerable. The concept is to flood the webserver with a very long *HttpServletRequest*, thereby making it crash.
There are two parameters, the first one is *req*, a *HttpServletRequest,* containing the request information like the parameters passed to the servlet etc. The second parameter is *res,* a *HttpServletResponse* object used to specify the HTTP header information etc.

8.2.5.2.29 public void doGet (HttpServletRequest req, HttpServletResponse res)

This method is called by the web server when a HTTP GET request is made. It simply calls the *processRequest* method (see 8.2.4.2.25). There is no risk for the “denial of service” attack described in 8.2.5.2.29 when GET is used to retrieve information from the server.
There are two parameters, the first one is *req*, a *HttpServletRequest,* containing the request information like the parameters passed to the servlet etc. The second parameter is *res,* a *HttpServletResponse* object used to specify the HTTP header information etc.

8.2.5.2.30 private void processRequest(HttpServletRequest req, HttpServletResponse res)

This method takes a look at the path information and parameters and performs the requested action. This is the core method of this servlet, as it decides which of the above methods should be called when. A lot of the above methods are not called from here though, but from other servlets (notably the *Management* Servlet, see 8.2.9).

There are two parameters, of which the first one is *req*, a *HttpServletRequest,* containing the request information like the parameters passed to the servlet etc. The second parameter is *res,* a *HttpServletResponse* object used to specify the HTTP header information etc.

8.2.5.2.31 public ServletOutputStream startOutput(HttpServletRequest req, HttpServletResponse res)

This method sets the HTTP content type and writes out the HTML header. It returns a *ServletOutputStream* object where the output of the servlet should be written to.
There are two parameters. The first one is *req*, a *HttpServletRequest,* containing the request information like the parameters passed to the servlet etc. The second parameter is *res,* a *HttpServletResponse* object used to specify the HTTP header information etc.

8.2.5.2.32 private void endOutput(ServletOutputStream out)

This method writes the end of the HTML document: it writes the </BODY> and </HTML> tags.
There is one parameter, *out*, of the type *ServletOutputStream*, containing the stream the output of the method is written to.

The code of the *Users* Servlet class can be found on the disk accompanying this report.

8.2.6 ProcessTransactions.java

ProcessTransactions.java is the servlet class that does all the interfacing with the hashtable that contains the transactions. Other servlets can only access the - private - hashtable *transactions* through methods of the *ProcessTransactions* class.

8.2.6.1 Variables

The *ProcessTransactions* class defines the following variables/constants:

- *private boolean debug = false;*
- *private static Hashtable transactions = new Hashtable();*
- *public static String filename;*
- *private static Enumeration eProcess = transactions.elements();*

For information about the *debug* variable, see 8.2.4.1.
The *transactions* hashtable holds all information about the products in memory; it is a hashtable with *Transaction* objects (see 8.2.3).
The String *filename* holds the filename of the database with transactions, as specified in the *servlets.properties* file. It is set in the *init()* method. For more information about the *servlets.properties* file refer to section 5.2.
The Enumeration *eProcess* is used in the *processProcessing()* method. It is used to store an Enumeration of Transaction objects that have not been processed. On subsequent calls, the *processProcessing()* method cycles through this Enumeration.

8.2.6.2 Methods

The *ProcessTransactions* class provides the following public methods:

- *void init(ServletConfig config)*
- *void destroy()*
- *void editTransaction(ServletOutputStream out, HttpServletRequest req)*
- *void listTransactions(ServletOutputStream out, HttpServletRequest req)*
- *void processProcessing(ServletOutputStream out, HttpServletRequest req, HttpServletResponse res)*
- *void processChange(ServletOutputStream out, HttpServletRequest req, HttpServletResponse res)*
- *void doSearch(ServletOutputStream out, HttpServletRequest req, boolean admin)*
- *void doPost (HttpServletRequest req, HttpServletResponse res)*
- *void doGet (HttpServletRequest req, HttpServletResponse res)*
- *ServletOutputStream startOutput(HttpServletRequest req, HttpServletResponse res)*

It also provides the following private methods:

- *void saveHashFile(String filename)*
- *int findHighestId()*
- *void readHashFile(String filename)*
- *void addToHash(Product p)*
- *void deleteFromHash(Product p)*
- *String addToPurchasedRights(String productId, int numberToAdd, StringBuffer oldPurchasedRights)*
- *String addTransaction(HttpServletRequest req, HttpSession mySession, Customer c)*
- *void processNewTransaction(HttpServletRequest req, HttpServletResponse res, ServletOutputStream out)*
- *void writeBeginningOfForm(ServletOutputStream out, String path, String formActio)*
- *void writeEndOfForm(ServletOutputStream out)*
- *void writeForm(ServletOutputStream out, Transaction t, String path, String formAction)*
- *static Transaction searchById(int searchId)*
- *void writeSearchButton(ServletOutputStream out, String path, String action)*
- *void writeSearchForm(ServletOutputStream out, String path)*
- *void processRequest(HttpServletRequest req, HttpServletResponse res)*
- *void endOutput(ServletOutputStream out)*

These methods are discussed in detail below.

8.2.6.2.1 public void init(ServletConfig config)

Called by the web server when the servlet is just loaded. In this method the class variables "debug" and "filename" are set, with values as specified in the servlet.properties file (see section 5.2). Then the *transactions hashtable* is read from file.
There is one parameter, *config*, of the type *ServletConfig*, which contains the configuration information and is passed to the *init* method by the server.

8.2.6.2.2 public void destroy(ServletConfig config)

Called by the web server when the server wants to drop the servlet from the JVM. Clears the hashtable in memory.
There is one parameter, *config*, of the type *ServletConfig*, which contains the configuration information and is passed to the *destroy* method by the server.

8.2.6.2.3 private void saveHashFile(String filename)

Called when the Hashtable has to be written to file. This method saves the *transactions* hashtable with one command, thanks to serialization!
There is one parameter, *filename*, of the type *String*, which contains the filename of the *products* file on disk.

8.2.6.2.4 private int findHighestId()

This method returns the highest Id in use in the *transactions* hashtable + 1. It is called from readHashFile(), because when readHashFile() is called, on load of the *transactions* database file from disk, we have to initialise the next available id for the Transaction class. Therefore, we have to find the highest Id in use in the hashtable, increase it with 1 and return it.

8.2.6.2.5 private void readHashFile(String filename)

Called when the Hashtable has to be read from file. Reads the complete hashtable with one command, thanks to serialization, and then sets the next available *Transaction* Id in the *Transaction* class by using the *findHighestId()* method.
There is one parameter, *filename*, of the type *String*, which contains the filename of the *products* file on disk.

8.2.6.2.6 private void addToHash(Transaction t)

This method adds a new transaction to, or updates a transaction in the *transactions* hashtable.
There is one parameter, *t*, of the type *Transaction*, which contains the *Transaction* object to be added or updated. The *id* field of the *Transaction* object is used as the key in the hashtable.

8.2.6.2.7 private void deleteFromHash(Transaction t)

This method deletes a product from the *transactions* hashtable.
There is one parameter, *t*, of the type *Transaction*, which contains the *Transaction* object to be deleted. The *id* field of the *Transaction* object is used as the key in the hashtable.

8.2.6.2.8 private String addToPurchasedRights(String productId, int numberToAdd, StringBuffer oldPurchasedRights)

This method adds a product to the *purchasedRights* of a *Customer* object. It checks if this product exists in the *purchasedRights*, and if it does, just changes the number of copies. Otherwise, it adds the product to the string. For more information on the format of the *purchasedRights* String, see section 8.2.2. The new *purchasedRights* String is passed to the calling method as the return value.

There are three parameters. The first one is *productId*, a *String* containing the id of the product to be added to the *purchasedRights String.* The second parameter is the *int numberToAdd*, which holds the number of purchased copies. The last parameter, *StringBuffer oldPurchasedRights*, holds the *purchasedRights* String (in the form of a StringBuffer of course) to be updated.

8.2.6.4.9 private String addTransaction(HttpServletRequest req, HttpSession mySession, Customer c)

This method adds transactions to the transactions hashtable. It processes the shopping cart contained in the mySession object passed as a parameter, and adds a transaction to the hashtable for every product that it contains. This method returns the new *purchasedRights* String for the Customer *c* passed as a parameter.

There are three parameters, of which the first one is *req*, a *HttpServletRequest,* containing the request information like the parameters passed to the servlet etc. The second parameter is *mySession*, of the type *HttpSession*. It holds the current Session object. The third parameter is *Customer c*, the customer initiating these transactions.

8.2.6.4.10 private void processNewTransaction(HttpServletRequest req, HttpServletResponse res, ServletOutputStream out)

The processing of the new transaction happens here. This method is called when the user clicks the final submit button, allowing this application to charge his creditcard. This method does all the necessary processing, calling the addTransaction() method, updating the customer record with the new purchasedRights String, and deleting the current shopping cart after the transaction.

There are three parameters. The first one is *req*, a *HttpServletRequest,* containing the request information like the parameters passed to the servlet etc. The second parameter is *res,* a *HttpServletResponse* object used to specify the HTTP header information etc. The third one is *out*, of the type *ServletOutputStream*, which is the stream the output of the method is written to.

8.2.6.2.11 private void writeBeginningOfForm(ServletOutputStream out, String path, String formAction)

This method writes the a form header with a form action as specified in the method parameters *formAction* and *path*. It also writes the <TABLE> tag.
There are three parameters. The first one is *ServletOutputStream out*, the stream the output of the method is written to. The second parameter is *path*, a *String* that contains the partial URL of the servlet (without the server name and without query-string). The third parameter is *formAction*, a *String* that is used to specify the form action.

8.2.6.2.12 private void writeEndOfForm(ServletOutputStream out)

This method writes the </FORM> tag to *ServletOutputStream out.*
There is one parameter, *out*, of the type *ServletOutputStream*, which is the stream the output of the method is written to.

8.2.6.2.13 private void writeForm(ServletOutputStream out, Transaction t, String path, String formAction)

This method is used to display a search or edit form, depending on the *formAction* parameter. New/Save/Delete buttons are also displayed.
There are four parameters, of which the first one is *ServletOutputStream out*, the stream the output of the method is written to. The second parameter is *t*, of the type *Transaction*, containing the *Transaction* object to be edited. The third parameter is *path*, a *String* that contains the partial URL of the servlet (without the server name and without query-string). The fourth parameter is *formAction*, a *String* that is used to specify the form action.

8.2.6.2.14 private static Transaction searchById(int searchId)

This method finds a transaction with matching id in the *transaction* hashtable. It returns the found *Transaction* object, or returns null if no match is found.
There is one parameter, *searchId*, an *int* containing the id of the transaction that is to be found.

8.2.6.2.15 public void editTransaction(ServletOutputStream out, HttpServletRequest req)

This method is called when a transaction needs to be edited by the administrator. Writes
a form with the fields of the transaction to be edited already filled in using the *writeForm* method,
and with save, new, and search buttons at the bottom. It also provides buttons to edit the *Customer* and the *Product* object of the transaction.
There are two parameters. The first one is *out*, of the type *ServletOutputStream*. It is the stream the output of the method is written to. The second parameter is *req*, a *HttpServletRequest,* that contains the request information like the parameters passed to the servlet etc.

8.2.6.2.16 public void listTransactions(ServletOutputStream out, HttpServletRequest req)

This method displays a list of available products, sorted alphabetically in the form "*p.name* by *c.lastName, c.firstName (t.date)*" . For an explanation of these fields, see section 8.2.1, 8.2.2 and 8.2.3. When the administrator makes a request for a list of transactions, this happens always through the *Management* servlet (see 8.2.9). A button is displayed that allows the creation of new *transactions*, and the links in the transaction list point to *editTransaction.*

There are two parameters, of which the first one is *out*, of the type *ServletOutputStream.* It is the stream the output of the method is written to. The second parameter is *req*, a *HttpServletRequest* that contains the request information like the parameters passed to the servlet etc.

8.2.6.2.18 private String getIdFromForm(HttpServletRequest req, String paramName)

Several forms use SELECT form objects with e.g. products or customers. The elements of the SELECT look like "id: name". This method retrieves the id from such a String. When no colon is found in the String passed to *getIdFromForm*, it simply returns the String "blank". The latter happens when the search form is called and the user leaves one of the SELECT fields blank.

There are two parameters. The first one is *req*, a *HttpServletRequest,* that contains the request information like the parameters passed to the servlet etc. The second parameter, the *String paramName*, is the name of the SELECT object to be found in the request information.

8.2.6.2.19 public void processProcessing(ServletOutputStream out, HttpServletRequest req, HttpServletResponse res)

This method creates a list of unprocessed transactions, and displays them to the administrator one at a time during each subsequent call to it. The administrator gets a form that shows him all information of an unprocessed transaction, allowing him to check a "processed" checkbox, and to change the number of copies, the amount of the transaction, and the currency of the transaction. The other fields can not be changed - if change in the other fields is necessary, one should edit the transaction (results in a call to the *editTransaction()* method). A "next" button at the bottom of the form allows to proceed to the next unprocessed transaction.

There are three parameters. The first one is *out*, of the type *ServletOutputStream*, which is the stream the output of the method is written to. The second parameter is *req*, a *HttpServletRequest,* containing the request information like the parameters passed to the servlet etc. The last one is *res,* a *HttpServletResponse* object used to specify the HTTP header information etc.

8.2.6.2.20 public void processChange(ServletOutputStream out, HttpServletRequest req, HttpServletResponse res)

If a change request occurs, this method processes the save (new or update) or delete request. It can obviously only be called by the administrator, through the *Management* Servlet (see section 8.2.9).
There are three parameters. The first one is *out*, of the type *ServletOutputStream.* It holds the stream the output of the method is written to. The second parameter is *req*, a *HttpServletRequest,* containing the request information like the parameters passed to the servlet etc. The last one is *res,* a *HttpServletResponse* object used to specify the HTTP header information etc.

8.2.6.2.21 private void writeSearchButton(ServletOutputStream out, String path, String action)

This method writes a html form with a search button and a set of 2 radiobuttons to *ServletOutputStream out*. The radiobuttons specify the type of search: AND or OR.
This method uses a parameter *action*, which can be either the *String* "Search" or the *String* "SearchNow". The *action* parameter equals "Search" when the search form must be displayed, and "SearchNow" when the form search results must be displayed.
There are three parameters. The first one is *ServletOutputStream out*, the stream the output of the method is written to. The second parameter is *path*, a *String* that contains the partial URL of the servlet (without the server name and without query-string). It is used to specify the form action. The third parameter is *action*, another *String*, whose function is explained higher.

8.2.6.2.22 private void writeSearchForm(ServletOutputStream out, String path)

This method displays a search form, allowing a search on customers and products.
There are two parameters. The first one is *ServletOutputStream out*, the stream the output of the method is written to. The second parameter is *path*, a *String* that contains the partial URL of the servlet (without the server name and without query-string). It is used to specify the form action.

8.2.6.2.23 public void doSearch(ServletOutputStream out, HttpServletRequest req)

If a search request occurs, this method processes the search. This means either display the search form, or do the search and display the results, depending on the action parameter of the pressed button (passed as a form variable via the *req HttpServletRequest*).
There are two parameters, of which the first one is *out*, of the type *ServletOutputStream.* It is the stream the output of the method is written to. The second parameter is *req*, a *HttpServletRequest,* containing the request information like the parameters passed to the servlet etc.

8.2.6.2.24 public void doPost (HttpServletRequest req, HttpServletResponse res

This method is called by the web server when a HTTP POST request is made. It checks if the length of the *HttpServletRequest req* is smaller than 8192 bytes; if it is, the *processRequest* method (see 8.2.6.2.26) is called. If not, the request is denied. This is to limit a typical "denial of service attack", to which some web servers are still vulnerable. The concept is to flood the webserver with a very long *HttpServletRequest*, thereby making it crash.
There are two parameters, of which the first one is *req*, a *HttpServletRequest* containing the request information like the parameters passed to the servlet etc. The second parameter is *res,* a *HttpServletResponse* object used to specify the HTTP header information etc.

8.2.6.2.25 public void doGet (HttpServletRequest req, HttpServletResponse res

This method is called by the web server when a HTTP GET request is made. It simply calls the *processRequest* method (see 8.2.4.2.25). There is no risk for the "denial of service" attack described in 8.2.6.2.24 when GET is used to retrieve information from the server.
There are two parameters. The first one is *req*, a *HttpServletRequest,* containing the request information like the parameters passed to the servlet etc. The second parameter is *res,* a *HttpServletResponse* object used to specify the HTTP header information etc.

8.2.6.2.26 private void processRequest(HttpServletRequest req, HttpServletResponse res)

This method takes a look at the path information and parameters and performs the requested action. This is the core method of this servlet, as it decides which of the above methods should be called when. A lot of the above methods are not called from here though, but from other servlets (notably the *Management* Servlet, see 8.2.9).
There are two parameters, of which the first one is *req*, a *HttpServletRequest,* containing the request information like the parameters passed to the servlet etc. The second parameter is *res,* a *HttpServletResponse* object used to specify the HTTP header information etc.

8.2.6.2.27 public ServletOutputStream startOutput(HttpServletRequest req, HttpServletResponse res)

This method sets the HTTP content type and writes out the HTML header. It returns a *ServletOutputStream* object where the output of the servlet should be written to.
There are two parameters. The first one is *req*, a *HttpServletRequest,* containing the request information like the parameters passed to the servlet etc. The second parameter is *res,* a *HttpServletResponse* object used to specify the HTTP header information etc.

8.2.6.2.28 private void endOutput(ServletOutputStream out)

This method writes the end of the HTML document: it writes the </BODY> and </HTML> tags.
There is one parameter, *out*, of the type *ServletOutputStream*, which is the stream the output of the method is written to.

The code of the *ProcessTransactions* Servlet class can be found on the disk accompanying this report.

8.2.7 Serve.java

Serve.java is the servlet class that serves files to the customers, provided they have access rights. It also includes a method to display a list of all products a particular customer has rights to, indicating how many copies he can view simultaneously.

8.2.7.1 Variables

The *Serve* class defines the following variables:

- *private boolean debug = false;*
- *private final String productAndUserKey = "userIdAndProductId";*
- *private final String driveLetter = "E:";*
- *private final String documentRoot = "/eCommDoc/";*

For information about the *debug* variable, see 8.2.4.1.
The String *productAndUserKey* is a final. This constant is merely a placeholder; it defines a string that is used as a key to save and retrieve a String containing a user Id and a product Id in a session. This String is used to limit the number of simultaneous logins to a site or document by a customer.
The finals *driveLetter* and *documentRoot* are used to create a hardcoded directory for all documents and sites that can be served by the *Serve* Servlet. They are prefixed to all files that have to be served, before the *realBaseDir* specified in the Product object. This has been done to avoid the following possible security risk. Without a hardcoded driveletter and documentRoot, a careless site administrator might leave the realBaseDir of a product blank, thus practically allowing access to all files on this hard disk for everyone who purchases the product. For more information on this problem, see section 7.3.7. The *documentRoot* String must always begin and end with a slash, and all slashes must be forward.

8.2.7.2 Methods

The *Serve* class provides the following public methods:

- *void init(ServletConfig config)*
- *void doPost (HttpServletRequest req, HttpServletResponse res)*
- *void doGet (HttpServletRequest req, HttpServletResponse res)*
- *ServletOutputStream startOutput(HttpServletRequest req, HttpServletResponse res)*

It also provides the following private methods:

- *String replaceSlashes(String filePath)*
- *void serveFile(HttpServletRequest req, HttpServletResponse res, Product p, String fileName)*
- *String extractAboveBaseDir(String vbdParam)*
- *int checkActiveProducts(HttpSessionContext thisContext, String hashtableKey)*
- *int checkPurchasedRights(Customer c, Product p)*
- *void handleFileRequest(HttpServletRequest req, HttpServletResponse res)*
- *void handleListRequest(HttpServletRequest req, HttpServletResponse res, ServletOutputStream out)*
- *void processRequest(HttpServletRequest req, HttpServletResponse res)*
- *void endOutput(ServletOutputStream out)*

These methods are discussed in detail below.

8.2.7.2.1 public void init(ServletConfig config)

Called by the web server when the servlet is just loaded. In this method the class variable "debug" is set, with the value as specified in the servlet.properties file (see section 4.2).
There is one parameter, *config*, of the type *ServletConfig*, which contains the configuration information and is passed to the *init* method by the server.

8.2.7.2.2 private String replaceSlashes(String filePath)

This method replaces all back slashes by forward slashes in the parameter *filePath*. It then returns the updated *String.* Called from serveFile.
The only parameter *filePath*, a *String*, is the String to parse.

8.2.7.2.3 private void serveFile(HttpServletRequest req, HttpServletResponse res, Product p, String fileName)

Here the actual serving of the files takes place. The file MIME type is determined depending on the file extension. Currently supported extensions are: HTML, SHTML, HTM, JPEG, JPG, GIF, ZIP, ARJ, GZ, TGZ and Z. All other extensions are served as being of the type "text/plain".
The path of the requested file is created from the virtualBaseDir. It is assured that by serving this file, no access is granted to a file above the document root of this product. When an error occurs, e.g. a fileNotFoundException, the user is informed. All files of the "text/html" type are parsed using the *ServeParser* class (see section 8.2.8). This method is called from the *handleFileRequest()* method.
There are four parameters. The first one is *req*, a *HttpServletRequest,* containing the request information like the parameters passed to the servlet etc. The second parameter is *res,* a *HttpServletResponse* object used to specify the HTTP header information etc. As the third parameter, *Product p* is passed. This product is the one with the *virtualBaseDir* matching the

one of the requested file. The last parameter is the *String fileName*, containing the name of the file to be served.

8.2.7.2.4 private String extractAboveBaseDir(String vbdParam)

This method returns everything after the first forward slash in the String *vbdParam* passed as a parameter. *extractAboveBaseDir()* is called from *handleFileRequest().*
The only parameter, the *String vbdParam*, holds the String to be parsed.

8.2.7.2.5 private int checkActiveProducts(HttpSessionContext thisContext, String hashtableKey)

This method checks how many sessions have the "productAndUserKey" set to the hashtableKey value supplied as a parameter. It returns that number. The *productAndUserKey* is a String containing the userId and productId, thus pinpointing who is viewing what document or site in a certain session. It is of course only used by the Serve Servlet, to check the number of concurrent logins to a certain document or site by a customer.
There are two parameters. The *HttpSessionContext thisContext* is used to retrieve all sessions, whereas the *String hashtableKey* holds the *productAndUserKey* to be searched for.

8.2.7.2.6 private int checkPurchasedRights(Customer c, Product p)

This method simply searches the purchasedRights String of Customer *c* and returns the number of simultaneous copies of Product *p* he is allowed to view. It is called from *handleFileRequest().*
The two parameters are *c,* the *Customer* object, and *p*, the *Product* object.

8.2.7.2.7 private void handleFileRequest(HttpServletRequest req, HttpServletResponse res)

If there is a valid session, this method checks the rights the user has to the requested file and handles accordingly. If there is a problem (like a request for too much simultaneous copies), the user is informed. When there are no problems, serveFile() is called to serve the file. If there is no valid session, the user is redirected to the login screen for returning customers.
There are two parameters. The first one is *req*, a *HttpServletRequest,* containing the request information like the parameters passed to the servlet etc. The second parameter is *res,* a *HttpServletResponse* object used to specify the HTTP header information etc.

8.2.7.2.8 private void handleListRequest(HttpServletRequest req, HttpServletResponse res, ServletOutputStream out)

If there is a valid session, this method checks which rights the user has and displays a list of the documents/sites where access will be granted, mentioning the number of simultaneous

accesses by the same customer that are allowed. If there is no valid session, the user is redirected to the login screen for returning customers.
There are three parameters, and the first one is *req*, a *HttpServletRequest,* containing the request information like the parameters passed to the servlet etc. The second parameter is *res,* a *HttpServletResponse* object used to specify the HTTP header information etc. The last parameter is *ServletOutputStream out*, the stream the output of the method is written to.

8.2.7.2.9 public void doPost (HttpServletRequest req, HttpServletResponse res)

This method is called by the web server when a HTTP POST request is made. It checks if the length of the *HttpServletRequest req* is smaller than 8192 bytes; if it is, the *processRequest* method (see 8.2.7.2.11) is called. If not, the request is denied. This is to limit a typical "denial of service attack", to which some web servers are still vulnerable. The concept is to flood the webserver with a very long *HttpServletRequest*, thereby making it crash.
There are two parameters, of which the first one is *req*, a *HttpServletRequest* containing the request information like the parameters passed to the servlet etc. The second parameter is *res,* a *HttpServletResponse* object used to specify the HTTP header information etc.

8.2.7.2.10 public void doGet (HttpServletRequest req, HttpServletResponse res)

This method is called by the web server when a HTTP GET request is made. It simply calls the *processRequest* method (see 8.2.7.2.11). There is no risk for the "denial of service" attack described in 8.2.7.2.9 when GET is used to retrieve information from the server.
There are two parameters. The first one is *req*, a *HttpServletRequest* containing the request information like the parameters passed to the servlet etc. The second parameter is *res,* a *HttpServletResponse* object used to specify the HTTP header information etc.

8.2.7.2.11 private void processRequest(HttpServletRequest req, HttpServletResponse res)

This method takes a look at the path information and parameters and performs the requested action. This is the core method of this servlet, as it decides which of the above methods should be called when.
There are two parameters. The first one is *req*, a *HttpServletRequest,* containing the request information like the parameters passed to the servlet etc. The second parameter is *res,* a *HttpServletResponse* object used to specify the HTTP header information etc.

8.2.7.2.12 public ServletOutputStream startOutput(HttpServletRequest req, HttpServletResponse res)

This method sets the HTTP content type and writes out the HTML header. It returns a *ServletOutputStream* object where the output of the servlet should be written to.

There are two parameters. The first one is *req*, a *HttpServletRequest,* containing the request information like the parameters passed to the servlet etc. The second parameter is *res,* a *HttpServletResponse* object used to specify the HTTP header information etc.

8.2.7.2.13 private void endOutput(ServletOutputStream out)

This method writes the end of the HTML document: it writes the </BODY> and </HTML> tags.
There is one parameter, *out*, of the type *ServletOutputStream*, which is the stream the output of the method is written to.

The code of the *Serve* Servlet class can be found on the disk accompanying this report.

8.2.8 ServeParser.java

ServeParser.java is a class that parses the html-files sent to the customers by the *Serve* Servlet, from where it is called.

8.2.8.1 Variables

The *ServeParser* class defines the following variables:

- *String prefix;*
- *int prefixLen;*
- *String fileName;*
- *int fileNameLen;*
- *final String quote = "\"";*
- *final String searchFor1 = "HREF=";   // links etc.*
- *final int len1 = 5;*
- *final String searchNotFor1a = "HREF=HTTP://";*
- *final String searchNotFor1b = "HREF=\"HTTP://";*
- *final String searchNotFor1c = "HREF=FTP://";*
- *final String searchNotFor1d = "HREF=\"FTP://";*
- *final String searchNotFor1e = "HREF=MAILTO:";*
- *final String searchNotFor1f = "HREF=\"MAILTO:";*
- *final String searchNotFor1g = "HREF=NEWS:";*
- *final String searchNotFor1h = "HREF=\"NEWS:";*
- *final String searchNotFor1i = "HREF=HTTPS://";*
- *final String searchNotFor1j = "HREF=\"HTTPS://";*
- *final String searchFor2 = "SRC=";   // img tags etc.*
- *final int len2 = 4;*
- *final String searchNotFor2a = "SRC=HTTP://";*
- *final String searchNotFor2b = "SRC=\"HTTP://";*

- *final String searchNotFor2c = "SRC=FTP:///";*
- *final String searchNotFor2d = "SRC=\"FTP://";*
- *final String searchNotFor2e = "SRC=HTTPS://";*
- *final String searchNotFor2f = "SRC=\"HTTPS://";*
- *final String searchFor3 = "CODEBASE=";   // applet tags etc.*
- *final int len3 = 9;*
- *final String searchNotFor3a = "CODEBASE=HTTP://";*
- *final String searchNotFor3b = "CODEBASE=\"HTTP://";*
- *final String searchNotFor3c = "CODEBASE=HTTPS://";*
- *final String searchNotFor3d = "CODEBASE=\"HTTPS://";*
- *final String searchFor4 = "BACKGROUND=";   // body tag etc.*
- *final int len4 = 11;*
- *final String searchNotFor4a = "BACKGROUND=HTTP://";*
- *final String searchNotFor4b = "BACKGROUND=\"HTTP://";*
- *final String searchNotFor4c = "BACKGROUND=HTTPS://";*
- *final String searchNotFor4d = "BACKGROUND=\"HTTPS://";*
- *int arrayLength, arrayLength2;*
- *byte[] b2 = new byte[13];*

The *prefix* and *fileName* Strings hold the Strings that should be prefixed. *prefixLen* and *fileNameLen*, both ints, respectively hold the length of the *prefix* and the *fileName* String. The *quote* String holds just a quote. The Strings with a name like *searchForX* with *X* a number contain the uppercase versions of the parameters to be searched for. The *lenX* ints hold the length of the corresponding *searchForX* String. These ints are defined to improve performance while executing the parse loops. The *searchNotForXx* Strings, with *X* a number and *x* a letter, hold the Strings that should be ignored when searching for the corresponding *searchForX* String. The *arrayLength* and *arrayLength2* ints are used in the *parsedRead()* method. They hold the length of the 2 buffers that are used to read in the bytes from the underlying *BufferedInputStream.* The byte array b2 is used to hold the “push-back” buffer (for more info see section 8.2.8.6) between subsequent calls to the *parsedRead()* method.

8.2.8.2 Methods

The *ServeParser* class provides the following public methods:

- *ServeParser(InputStream in, int size, String prefix)*
- *final String parsedRead(int len)*

It also provides the following private methods:

- *void searchForHREF(String tmpStr, StringBuffer tmpStrBuff)*
- *void searchForSRC(String tmpStr, StringBuffer tmpStrBuff)*
- *void searchForCODEBASE(String tmpStr, StringBuffer tmpStrBuff)*
- *void searchForBACKGROUND(String tmpStr, StringBuffer tmpStrBuff)*

In addition to these methods, *ServeParser* also defines an inner class, *public static class Test*, to make testing easier.

These methods are discussed in detail below.

8.2.8.2.1 public ServeParser(InputStream in, int size, String prefix)

This is the constructor for the class. It initialises the class variables *prefix*, *prefixLen*, *fileName* and *fileNameLen*. The parameter *prefix* is split in 2 parts, *fileName* is initialised to the part after the last slash, and class variable *prefix* to the part before (and including) that slash. This method obviously also calls the constructor of the super class.
There are three parameters. *inputStream in* is the inputStream where the bytes will be read from. The *int size* determines the buffer size, and the *String prefix* holds the maximal String to be prefixed.

8.2.8.2.2 private void searchForHREF(String tmpStr, StringBuffer tmpStrBuff) {

This method searches the String *tmpStr* (passed as a parameter) for the HREF tag parameter, as specified in the class variable *searchFor1*. It ignores the HREFs matching the class variables *searchNotFor1x*, with *x* from a to j. When there is a match, a prefix is inserted, either just the class variable *prefix*, or *prefix* and the class variable *filename* in the case of an anchor (see section 7.3.7 for more information). The String to be parsed is *tmpStr*, and the result is returned in StringBuffer *tmpStrBuff*.

8.2.8.2.3 private void searchForSRC(String tmpStr, StringBuffer tmpStrBuff)

This method searches the String *tmpStr* (passed as a parameter) for the SRC tag parameter, as specified in the class variable *searchFor2*. It ignores the SRCs matching the class variables *searchNotFor1x*, with *x* from a to f. When there is a match, a prefix is inserted, either just the class variable *prefix*, or *prefix* and the class variable *filename* in the case of an anchor (see section 7.3.7 for more information). The String to be parsed is *tmpStr*, and the result is returned in StringBuffer *tmpStrBuff*.

8.2.8.2.4 private void searchForCODEBASE(String tmpStr, StringBuffer tmpStrBuff)

This method searches the String *tmpStr* (passed as a parameter) for the CODEBASE tag parameter, as specified in the class variable *searchFor3*. It ignores the CODEBASEs matching the class variables *searchNotFor3x*, with *x* from a to d. When there is a match, a prefix is inserted, either just the class variable *prefix*, or *prefix* and the class variable *filename* in the case of an anchor (see section 7.3.7 for more information). The String to be parsed is *tmpStr*, and the result is returned in StringBuffer *tmpStrBuff*.

8.2.8.2.5 private void searchForBACKGROUND(String tmpStr, StringBuffer tmpStrBuff)

This method searches the String *tmpStr* (passed as a parameter) for the BACKGROUND tag parameter, as specified in the class variable *searchFor4*. It ignores the BACKGROUNDs matching the class variables *searchNotFor4x*, with *x* from a to d. When there is a match, a prefix is inserted, either just the class variable *prefix*, or *prefix* and the class variable *filename* in the case of an anchor (see section 7.3.7 for more information). The String to be parsed is *tmpStr*, and the result is returned in StringBuffer *tmpStrBuff*.

8.2.8.2.6 public final String parsedRead(int len)

The actual reading from the underlying buffered stream happens here. A buffer with the size specified in the parameter *len* - 13 is read, and then a second buffer is read of 13 bytes. This second buffer is simply appended to the first buffer, and stored in the class variable *b2*. When the end of the total buffer is reached, the parsed buffer is returned, but without the last 13 characters. They are put in front of the buffer at the next read. The reason for this extra "pushback" buffer is that when the buffer is parsed, one of the Strings to be searched for might be cut in 2 parts by the end of the buffer. Without the 13 byte pushback buffer, this String would simply not be found.
The length of the pushback buffer is 13 bytes because the longest String to search for is 'BACKGROUND=', 11 bytes long. In addition to this, we need access to the 2 following bytes (11+2 = 13), as they might be a quote and a hash. So, if we add the 13 bytes to the read buffer, but not react on any Strings found closer than 13 bytes to the end of the total buffer (as these will be put in front of the buffer on the next read), everything will be parsed correctly.
The *int len* is the only parameter, specifying the number of bytes to be read.

The code of the *ServeParser* Servlet class can be found on the disk accompanying this report.

8.2.9 Management.java

Management.java is a servlet class that provides only the authentication and the calling of the management-methods from the Products, Users and Transactions servlets. These methods can not be called directly from the web from these servlets.

8.2.9.1 Variables

The *Management* class defines the following variables:

- *private boolean debug = false;*
- *private String adminLogin = "test";*
- *private String adminPassword = "test";*
- *private final String redirectParam = "redirectDestination";*

- *private static StringBuffer currentManagerId = new StringBuffer();*
- *public static final String manageParam = "Manager";	// also accessed from Users servlet*

For information about the *debug* variable, see 8.2.4.1.
The *adminLogin* and *adminPassword* Strings are the login credentials that are necessary to log in as administrator of the site.
The String *redirectParam* is a final. This constant is merely a placeholder; it defines a string that is used as a key to save and retrieve a String containing the redirect destination in a session. This String is used when a page is requested but another one must first be passed, e.g. a login screen.
The StringBuffer *currentManagerId* holds the Id of the current administration session, or null if there is none. It is used as an extra security to make sure that a management session can not be forged.
The String manage*Param* is also a final. This constant is merely a placeholder as well; it defines a string that is used as a key to save and retrieve a String that indicates that a session is the management session.

8.2.9.2 Methods

The *Management* class provides the following public methods:

- *void init(ServletConfig config)*
- *static String getServletDir(String path)*
- *static boolean isManagementSession(HttpServletRequest req)*
- *void doPost (HttpServletRequest req, HttpServletResponse res)*
- *void doGet (HttpServletRequest req, HttpServletResponse res)*
- *ServletOutputStream startOutput(HttpServletRequest req, HttpServletResponse res)*

It also provides the following private methods:

- *void askManagementPassword(HttpServletRequest req, ServletOutputStream out)*
- *void handleProducts(HttpServletRequest req, ServletOutputStream out)*
- *void handleTransactions(HttpServletRequest req, HttpServletResponse res, ServletOutputStream out)*
- *void handleUsers(HttpServletRequest req, HttpServletResponse res, ServletOutputStream out)*
- *void logoutManagement(HttpServletRequest req, ServletOutputStream out, HttpSession mySession)*
- *String activeManagementSession(HttpSessionContext thisContext)*
- *void checkManagementLogin(HttpServletRequest req, HttpServletResponse res)*
- *void processRequest(HttpServletRequest req, HttpServletResponse res)*
- *void endOutput(ServletOutputStream out)*

These methods are discussed in detail below.

8.2.9.2.1 public void init(ServletConfig config)

Called by the web server when the servlet is just loaded. In this method the class variable "debug" is set, with the value as specified in the servlet.properties file (see section 4.2).
There is one parameter, *config*, of the type *ServletConfig*, which contains the configuration information and is passed to the *init* method by the server.

8.2.9.2.2 public static String getServletDir(String path)

This method extracts the virtual directory of the servlet out of the URL. For example, when the management servlet would be called as *http://some.server.name/servlets/Management*, this method would return *servlets/ .*
There is one parameter, *path*, of the type *String*, which contains the partial URL of the servlet (without the server name and without query-string).

8.2.9.2.3 private void askManagementPassword(HttpServletRequest req, ServletOutputStream out)

This method is called when a user wants to enter the management-side of the site. This function merely displays the login form for the administrator. The information sent when the login button is pushed is not encrypted, so this function should never be called over a non-SSL connection!!
There are two parameters. The first one is *req*, a *HttpServletRequest,* containing the request information like the parameters passed to the servlet etc. The second parameter, *ServletOutputStream out*, is the stream the output of the method is written to.

8.2.9.2.4 private void handleProducts(HttpServletRequest req, ServletOutputStream out)

This method writes several links at the top of the screen. It reacts on URLs like http:/some.server.name/servlet/Management/products/... If nothing is specified after "products" in the URL, it just asks the administrator to make his choice from the above links. The links provided are: main management page, user management page and transaction management page. Also links to list products, add products and search products are provided. If any of these last three links is clicked, this method reacts again, calling the appropriate method in the Products Servlet, writing its output under the line of links and the title of the page.
There are two parameters. The first one is *req*, a *HttpServletRequest*, containing the request information like the parameters passed to the servlet etc. The second parameter, *ServletOutputStream out*, is the stream the output of the method is written to.

8.2.9.2.5 private void handleTransactions(HttpServletRequest req, HttpServletResponse res, ServletOutputStream out)

This method writes several links at the top of the screen. It reacts on URLs like http:/some.server.name/servlet/Management/transactions/... If nothing is specified after "transactions" in the URL, it just asks the administrator to make his choice from the above links. The links provided are: main management page, product management page and user

management page. Also links to list transactions, process transactions and search transactions are provided. If any of these last links is clicked, this method reacts again, calling the appropriate method in the ProcessTransactions Servlet, writing its output under the line of links and the title of the page.
There are two parameters. The first one is *req*, a *HttpServletRequest*, containing the request information like the parameters passed to the servlet etc. The second parameter, *ServletOutputStream out*, is the stream the output of the method is written to.

8.2.9.2.6 private void handleUsers(HttpServletRequest req, HttpServletResponse res, ServletOutputStream out)

This method writes several links at the top of the screen. It reacts on URLs like http:/some.server.name/servlet/Management/users/... If nothing is specified after "users" in the URL, it just ask the administrator to make his choice from the above links. The links provided are: main management page, product management page and transaction management page. Also links to list users, add users and search users are provided. If any of these last links is clicked, this method reacts again, calling the appropriate method in the Users Servlet, writing its output under the line of links and the title of the page.
There are three parameters. The first one is *req*, a *HttpServletRequest*, containing the request information like the parameters passed to the servlet etc. The second parameter is *res,* a *HttpServletResponse* object used to specify the HTTP header information etc. The third parameter, *ServletOutputStream out*, is the stream the output of the method is written to.

8.2.9.2.7 private void logoutManagement(HttpServletRequest req, ServletOutputStream out, HttpSession mySession)

This method logs out the administrator, and provides a link to the main shop page. If the administrator was not logged in, this is mentioned and the same link is provided.
There are three parameters. The first one is *req*, a *HttpServletRequest*, containing the request information like the parameters passed to the servlet etc. The second parameter, *ServletOutputStream out*, is the stream the output of the method is written to. The last parameter is *mySession,* of the type *HttpSession*, and holds the current session.

8.2.9.2.8 public static boolean isManagementSession(HttpServletRequest req)

This method checks if the current session is the Management session or not. If so, it returns true, otherwise, it returns false. This method is called from the *Users* Servlet, methods *processNewCustomer()* and *processRequest()*.
The only parameter is *req*, a *HttpServletRequest*, containing the request information like the parameters passed to the servlet etc.

8.2.9.2.9 private String activeManagementSession(HttpSessionContext thisContext)

This method finds a session with management parameter set. If there is none, it returns null, otherwise, it returns the session Id.
The parameter *HttpSessionContext thisContext* is used to retrieve all sessions.

8.2.9.2.10 private void checkManagementLogin(HttpServletRequest req, HttpServletResponse res)

This method checks if the login credentials provided are correct. If so, it logs out the user if there was one connected to the Session, and connects this session to the Administrator. Furthermore, for extra security, it sets the static class variable currentManagerId to the session id of this session. Then it redirects to the redirect destination stored in the session, or to the main management page if there was none.
There are two parameters. The first one is *req*, a *HttpServletRequest*, containing the request information like the parameters passed to the servlet etc. The second parameter, *res,* a *HttpServletResponse,* is used to specify the HTTP header information etc.

8.2.9.2.11 public void doPost (HttpServletRequest req, HttpServletResponse res)

This method is called by the web server when a HTTP POST request is made. It checks if the length of the *HttpServletRequest req* is smaller than 8192 bytes; if it is, the *processRequest* method (see 8.2.9.2.13) is called. If not, the request is denied. This is to limit a typical “denial of service attack”, to which some web servers are still vulnerable. The concept is to flood the webserver with a very long *HttpServletRequest*, thereby making it crash.
There are two parameters. The first one is *req*, a *HttpServletRequest,* containing the request information like the parameters passed to the servlet etc. The second parameter is *res,* a *HttpServletResponse* object used to specify the HTTP header information etc.

8.2.9.2.12 public void doGet (HttpServletRequest req, HttpServletResponse res)

This method is called by the web server when a HTTP GET request is made. It simply calls the *processRequest* method (see 8.2.9.2.13). There is no risk for the “denial of service” attack described in 8.2.9.2.11 when GET is used to retrieve information from the server.
There are two parameters, of which the first one is *req*, a *HttpServletRequest* containing the request information like the parameters passed to the servlet etc. The second parameter is *res,* a *HttpServletResponse* object used to specify the HTTP header information etc.

8.2.9.2.13 private void processRequest(HttpServletRequest req, HttpServletResponse res)

This method takes a look at the path information and parameters and performs the requested action. This is the core method of this servlet, as it decides which of the above methods should be called when.

There are two parameters. The first one is *req*, a *HttpServletRequest,* containing the request information like the parameters passed to the servlet etc. The second parameter is *res,* a *HttpServletResponse* object used to specify the HTTP header information etc.

8.2.9.2.14 public ServletOutputStream startOutput(HttpServletRequest req, HttpServletResponse res)

This method sets the HTTP content type and writes out the HTML header. It returns a *ServletOutputStream* object where the output of the servlet should be written to.
There are two parameters. The first one is *req*, a *HttpServletRequest,* containing the request information like the parameters passed to the servlet etc. The second parameter is *res,* a *HttpServletResponse* object used to specify the HTTP header information etc.

8.2.9.2.15 private void endOutput(ServletOutputStream out)

This method writes the end of the HTML document: it writes the </BODY> and </HTML> tags.
There is one parameter, *out*, of the type *ServletOutputStream*, which is the stream the output of the method is written to.

The code of the *Management* Servlet class can be found on the disk accompanying this report.

8.2.10 Cart.java

Cart.java is the servlet class that does all the interfacing with the shopping cart saved in the session.

8.2.10.1 Variables

The *Cart* class defines the following variables:

- *private boolean debug = false;*
- *public final String cartVarName = "cartContents";*

For information about the *debug* variable, see 8.2.4.1.
The String *cartVarName* is a final. This constant is merely a placeholder; it defines a string that is used as a key to save and retrieve the shopping cart contents String in a session.

8.2.10.2 Methods

The *Cart* class provides the following public methods:

- *void init(ServletConfig config)*
- *void viewCart(HttpServletRequest req, ServletOutputStream out, boolean buying)*
- *void doPost (HttpServletRequest req, HttpServletResponse res)*
- *void doGet (HttpServletRequest req, HttpServletResponse res)*

- *ServletOutputStream startOutput(HttpServletRequest req, HttpServletResponse res)*

It also provides the following private methods:

- *void String addToUnEmptyCart(HttpServletRequest req, StringBuffer oldValue)*
- *void addToCart(HttpServletRequest req, HttpServletResponse res, ServletOutputStream out)*
- *void updateCart(HttpServletRequest req, HttpServletResponse res, ServletOutputStream out)*
- *void writeBeginningOfForm(ServletOutputStream out, String action)*
- *void listCartContents(HttpServletRequest req, ServletOutputStream out, String oldCartValue, boolean buying)*
- *void writeViewCartButtons(ServletOutputStream out)*
- *void writeBuyCartButtons(ServletOutputStream out)*
- *String deleteFromCart(HttpServletRequest req, StringBuffer oldValue)*
- *void deleteCart(HttpServletRequest req, HttpServletResponse res, ServletOutputStream out)*
- *void processRequest(HttpServletRequest req, HttpServletResponse res)*
- *void endOutput(ServletOutputStream out)*

These methods are discussed in detail below.

8.2.10.2.1 public void init(ServletConfig config)

Called by the web server when the servlet is just loaded. In this method the class variable "debug" is set, with the value as specified in the servlet.properties file (see section 4.2).
There is one parameter, *config*, of the type *ServletConfig*, which contains the configuration information and is passed to the *init* method by the server.

8.2.10.2.2 private String addToUnEmptyCart(HttpServletRequest req, StringBuffer oldValue)

This method adds a product to the shopping cart in this session. It checks if this product exists in the cart, and if it does, just increases the number of copies. Otherwise, it adds the product to the string. The format of the cart string is the same as the format of the purchasedRights String in the Customer class: product=copies&product=copies&... The new cart contents String is passed to the calling method (*addToCart*) as the return value.
There are two parameters, of which the first one is *req*, a *HttpServletRequest,* containing the request information like the parameters passed to the servlet etc. The second parameter is the *StringBuffer oldValue*, containing the original cart contents.

8.2.10.2.3 private void addToCart(HttpServletRequest req, HttpServletResponse res, ServletOutputStream out)

The adding of a product to the shopping cart happens here. This method is called when the user clicks on the “add to cart” link when viewing a product’s details. If the cart was empty, a new cart contents String is created in the session by this method; otherwise, the *addToUnEmptyCart()* method is called to do the updating of the cart.
There are three parameters. The first one is *req*, a *HttpServletRequest,* containing the request information like the parameters passed to the servlet etc. The second parameter is

res, a *HttpServletResponse* object used to specify the HTTP header information etc. The last parameter is *ServletOutputStream out*, the stream the output of the method is written to.

8.2.10.2.4 private void updateCart(HttpServletRequest req, HttpServletResponse res, ServletOutputStream out)

The updating of the shopping cart happens here. When the customer views his cart contents, he has the possibility to change the number of copies of products in the cart. He can not add products from that form, but he can delete products by changing the number of copies to 0. When the submit button on the bottom of the view cart form is pressed, the method is called to process all the form parameters and update the cart contents.

There are three parameters. The first one is *req*, a *HttpServletRequest,* containing the request information like the parameters passed to the servlet etc. The second parameter is *res,* a *HttpServletResponse* object used to specify the HTTP header information etc. The last parameter is *ServletOutputStream out*, the stream the output of the method is written to.

8.2.10.2.5 private void writeBeginningOfForm(ServletOutputStream out, String action)

This method writes the a form header with a form action as specified in the method parameter *formAction*. It also writes the <TABLE> tag.
There are two parameters. The first one is *ServletOutputStream out*, the stream the output of the method is written to. The last parameter is *formAction*, a *String* that is used to specify the form action.

8.2.10.2.6 private void listCartContents(HttpServletRequest req, ServletOutputStream out, String oldCartValue, boolean buying)

Depending on the *buying* parameter, this method displays an “edit Cart” form or a “buy Cart” form. The “edit Cart” form allows the customer to change his cart contents. He can remove products by setting the number of copies to 0, or just change the number of copies. Other products can not be added from this form. The *Submit Changes* button at the bottom of the form is connected to the *updateCart()* method through the *processRequest()* method. Pressing the second button, *Buy Now*, will result in the authentication of the user, and after that, the display of the “buy Cart” form. This form merely shows the cart contents, with no way to edit them, and supplies a *Submit* button with a warning that the customer’s credit card will be charged with the amount specified higher on the form if he presses it.
There are three parameters. The first one is *req*, a *HttpServletRequest,* containing the request information like the parameters passed to the servlet etc. The second parameter is *out*, of the type *ServletOutputStream*, which is the stream the output of the method is written to. The last parameter, the *boolean buying*, is explained higher.

8.2.10.2.7 private void writeViewCartButtons(ServletOutputStream out)

This method writes 2 buttons at the bottom of the view Cart form. The first one is "Submit Changes", and simply causes the updateCart() method to be called to recalculate the cart. The second button is "Buy Now", and will result in the authentication of the customer and after that the displaying of the "buy Cart" form through the *viewCart()* and *listCartContents()* methods.
The only parameter *out*, of the type *ServletOutputStream*, is the stream the output of the method is written to.

8.2.10.2.8 private void writeBuyCartButtons(ServletOutputStream out)

This method writes a "Submit" button at the bottom of the buy Cart form.
The only parameter *out*, of the type *ServletOutputStream*, is the stream the output of the method is written to.

8.2.10.2.9 public void viewCart(HttpServletRequest req, ServletOutputStream out, boolean buying)

This method displays the shopping cart by calling the listCartContents() method. Depending on the "buying" parameter, the cart is editable or not, and the buttons at the bottom of the form are different. From here, *listCartContents()* is called to do the actual displaying of the cart.
There are three parameters. The first one is *req*, a *HttpServletRequest,* containing the request information like the parameters passed to the servlet etc. The second parameter is *out*, of the type *ServletOutputStream*, which is the stream the output of the method is written to. The last parameter, the *boolean buying*, is explained in section 8.2.10.2.6, *listCartContents()*.

8.2.10.2.10 private String deleteFromCart(HttpServletRequest req, StringBuffer oldValue)

This method allows the deletion of a product from the shopping cart. It is no longer used in this application, but works perfectly. This method is called from the deleteCart() method. It is still present in this sourcecode because it might come in handy in the future.
There are two parameters, of which the first one is *req*, a *HttpServletRequest,* containing the request information like the parameters passed to the servlet etc. The second parameter is the *StringBuffer oldValue*, containing the original cart contents.

8.2.10.2.11 private void deleteCart(HttpServletRequest req, HttpServletResponse res, ServletOutputStream out)

This method allows the deletion of a product from the shopping cart, or the deletion of the entire cart if no product id is specified. It calls the deleteFromCart() method in the first case. This method is no longer used in this application, but works perfectly. It is still present in this sourcecode because it might come in handy in the future.
There are three parameters. The first one is *req*, a *HttpServletRequest,* containing the request information like the parameters passed to the servlet etc. The second parameter is

res, a *HttpServletResponse* object used to specify the HTTP header information etc. The last parameter is *ServletOutputStream out*, the stream the output of the method is written to.

8.2.10.2.12 public void doPost (HttpServletRequest req, HttpServletResponse res)

This method is called by the web server when a HTTP POST request is made. It checks if the length of the *HttpServletRequest req* is smaller than 8192 bytes; if it is, the *processRequest* method (see 8.2.10.2.14) is called. If not, the request is denied. This is to limit a typical "denial of service attack", to which some web servers are still vulnerable. The concept is to flood the webserver with a very long *HttpServletRequest*, thereby making it crash.
There are two parameters, of which the first one is *req*, a *HttpServletRequest* containing the request information like the parameters passed to the servlet etc. The second parameter is *res,* a *HttpServletResponse* object used to specify the HTTP header information etc.

8.2.10.2.13 public void doGet (HttpServletRequest req, HttpServletResponse res)

This method is called by the web server when a HTTP GET request is made. It simply calls the *processRequest* method (see 8.2.10.2.14). There is no risk for the "denial of service" attack described in 8.2.10.2.12 when GET is used to retrieve information from the server.
There are two parameters. The first one is *req*, a *HttpServletRequest,* containing the request information like the parameters passed to the servlet etc. The second parameter is *res,* a *HttpServletResponse* object used to specify the HTTP header information etc.

8.2.10.2.14 private void processRequest(HttpServletRequest req, HttpServletResponse res)

This method takes a look at the path information and parameters and performs the requested action. This is the core method of this servlet, as it decides which of the above methods should be called when. Some of the above methods are not called from here though, but from other servlets (notably the *Management* Servlet, see 8.2.9).
There are two parameters. The first one is *req*, a *HttpServletRequest,* containing the request information like the parameters passed to the servlet etc. The second parameter is *res,* a *HttpServletResponse* object used to specify the HTTP header information etc.

8.2.10.2.15 public ServletOutputStream startOutput(HttpServletRequest req, HttpServletResponse res)

This method sets the HTTP content type and writes out the HTML header. It returns a *ServletOutputStream* object where the output of the servlet should be written to.
There are two parameters, of which the first one is *req*, a *HttpServletRequest,* containing the request information like the parameters passed to the servlet etc. The second parameter is *res,* a *HttpServletResponse* object used to specify the HTTP header information etc.

8.2.10.2.16 private void endOutput(ServletOutputStream out)

This method writes the end of the HTML document: it writes the </BODY> and </HTML> tags.
There is one parameter, *out*, of the type *ServletOutputStream*, which is the stream the output of the method is written to.

The code of the *Cart* Servlet class can be found on the disk accompanying this report.

8.2.11 CheckCC.java

CheckCC.java is the credit card validation class that provides the methods to check the validity of a credit card number. These Credit Card Validation functions come from Netscape Communications Corporation FormChek.js (JavaScript 1.0 version), available at http://developer.netscape.com/docs/examples/javascript/formval/overview.html
The original version was in Javascript, and I translated the JavaScript to Java. Because I did not write these methods myself, I will not describe them separately. For more information on each method, see the source code of this class.

The code of the *CheckCC* class can be found on the disk accompanying this report.

8.2.12 Sorter.java

This is a sorter class, copied from "Java Examples in a Nutshell". (http://www.oreilly.com/catalog/books/jenut/). Copyright (c) 1997 by David Flanagan.
The following statements are at the top of the class:

> *This example is provided WITHOUT ANY WARRANTY either expressed or implied.*
> *You may study, use, modify, and distribute it for non-commercial purposes.*
> *For any commercial use, see http://www.davidflanagan.com/javaexamples*

If this site is to be used for commercial purposes, either this sorter class should be removed (all that needs to be written then is a sorting algorithm for Strings - it is used in the *Products, Users* and *ProcessTransaction* Servlets). Alternatively, all the examples could be licensed for a quite reasonable price: 50 $ per programmer using the examples for commercial use, of 500 $ for a site license for any number of programmers within an organisation.
If no license is taken, there is another piece of code that should be modified: the part that sends an e-mail to a customer when he forgets his password, in the *Users* servlet, comes also originally from one or the examples in this book - albeit changed to fit the purpose it serves here.

The code of the *Sorter* class can be found on the disk accompanying this report.

8.2.13 CustomerDetailsFormCheck.js

This the JavaScript source code used to do the checking of form-fields on the client, when a customer edits his details, and when an existing user logs in.

I will only describe the four function that I wrote, the others come from Netscape Communications Corporation FormChek.js (JavaScript 1.0 version), available at http://developer.netscape.com/docs/examples/javascript/formval/overview.html

8.2.13.1 function checkCC(form_element)

This function is called from the onChange event in the creditCardNumber text field on the new or edit user form. It calls the *isCardMatch* function to check the validity of the provided number. The *form_element* passed as an argument is the creditCardNumber text field.

8.2.13.2 function shortCheckFields(form)

This function gets called when the submit button is pressed on the login form for existing users. It checks whether the *emailAddress* field is a valid e-mail address and whether the *password* field is not blank. If any of these criteria is not met, an error message is displayed and the form is not submitted.
The *form* parameter holds the form object.

8.2.13.3 function checkFields(form)

This function is called when new users have filled out the complete new or edit user form. It checks if fields are blank, if the provided e-mail address is of the right form, if the two passwords match,... If any of these criteria is not met, an error message is displayed and the form is not submitted.
The *form* parameter holds the form object.

8.2.13.4 function fillInCardholderName(form)

This function is called in the onChange event of the lastName field on the new or edit user form. It fills in the combination *firstName*+" "+*lastName* as default for the *creditCardName* field.
The *form* parameter holds the form object.

The code of the CustomerDetailsFormCheck.js file can be found on the disk accompanying this report.

9. Installing the software on a computer

There are several necessary steps. First of all, obtain a copy a web server of your choice that supports servlets. If you do not use Netscape Enterprise Server (v3.5.1 or higher), chances are you will not need a copy of JRun (Live Software, v2.1.2 or higher), as I did not use JRun specific java code. However, I have not tried this. If you do use NES, get the correct copy of JRun (v2.1.2 or higher). In the rest of this section, I will assume that you use the combination of NES and JRun.

Start by installing NES. The installation is straightforward, and more information about the configuration can be found in appendix 11.1.3.

Next, install JRun. Here, too, the installation is straightforward, just follow the instructions of the installation program. You will have to make some changes to the obj.conf file in the configuration directory of the web server (typically "\Netscape\SuiteSpot\https-ServerName\config"), but these changes are documented very well in the documentation that comes with JRun.

Then copy the class files of the servlets of this thesis project to the directory that you have chosen to be your servlet directory (as specified in the obj.conf file). The class files (java bytecode) will work on any platform without recompilation. If you want to make changes to the software, you will have to copy the java source files as well. In that case, you will also have to install the Java Development Kit, which contains the *javac* java compiler. Install version 1.2 Beta 3 or higher - the source code will not compile with an older version of the JDK.

Make your changes using a normal text editor (make sure that it supports long filenames!), and when they have been made, recompile the java code by typing '*javac <filename>*' on the command line, where <filename> is the name of the file, including the *.java* extension. Be sure to type the exact filename (javac is case-sensitive!). When the source code compiles without errors, restart the secure server (on a Windows system: go to the *Services* section in the *Control Panel*, and stop and restart the correct NES server). At that point, you can access the site through a web-browser and see the changes you have made.

Finally, copy the files *general.css* (the Cascading Style Sheets file used for some displaying of forms) and *CustomerDetailsFormCheck.js* (the Javascript file with the code for the checking of form fields on the client) to the document root of the web server, typically \Netscape\SuiteSpot\docs.

10. Conclusion and further work

The program works entirely, with no known bugs. The requested functionality is entirely available: the secure purchase of items over the Internet, featuring shopping carts and Credit Card validation, and featuring a customer and an administration interface. An extra advantage of the approach I have chosen (the development of the whole site in Java Servlets) is the portability of the solution. It will run on any platform with a web server that supports Java Servlets or that supports the JRun plugin.

This program will work satisfactory for a small business with not too much customers. For big-scale deployment, a major change will have to be made: the *Products, Users* and *ProcessTransactions* Servlets should be changed in such a way that they no longer use a internal hashtable to store their data. The data should then be stored in an “industrial strength” database like DB2, Oracle or Sybase, and accessed via Java Database Connectivity (JDBC).

A few other enhancements could be made. For instance, multiple currency support could be built in quite easily. Also interesting would be the possibility to purchase time-limited access to a document or site. Those two feature could be added quite easily, because for both of them the foundations have been provided: every product record has a *currency* field, and a *dateAvailable* and a *dateExpired* field. These three fields are accessible through the administration interface.

Another enhancement could be the automation of the transaction processing. Currently, an operator has to check a *processed* checkbox on every new transaction, when he processes it. It would be conceivable to feed the transaction information directly into special programs from the credit card companies, and thus automating the transaction processing for all transactions but these that have something special.

There are a lot more improvements that can be made, but all together, this implementation is already a quite usable solution for the electronic purchase of access rights to documents or websites.

11. Appendices

11.1 Reviews and documentation

11.1.1 Installing Novell Netware 4.11 Server

1. Hardware requirements:
    - If you want to make your server boot without diskettes, you will need a DOS-partition of at least 15MB. It is recommended to make it 1MB bigger for every MB of RAM you have.
    - RAM: 20MB + 0.008 x disc space + 1 to 4 MB cache buffer RAM (recommended for performance)
1. The installation can be done from a CD, or from the network, using another NW 4.11 machine. I'll describe the local CD-installation.
2. What else do we need?
    - Novell Netware 4.11 Installation CDs (4)
    - A bootable DOS or Win95 disc
    - A disc with the drivers for the CD-ROM drive
1. The actual installation begins with calculating the necessary amount of space on the DOS-partition. As I install on a machine with 32MB RAM, the partition should be 15 + 32 x 1 = 47 MB big. If you want to add more software to it later, you make it bigger. I chose 100 MB, just in case.
2. Calculate the necessary amount of RAM: 20 MB + 0.008 x 1.6 GB + 1 to 4 MB Cache Buffer RAM would amount to 33.8 to 36.8 MB. But as this server will initially only be used for authentication, 32 MB will suffice. We can always add RAM later.
3. Now create the (bootable!) DOS-partition using FDISK and FORMAT. After that, install the drivers for the CD-ROM.
4. Insert the first NW 4.11 CD, and start the "install" program that can be found under the root.
    - After the program starts, choose the installation language. Then read the license agreement, pressing *enter* four times during this process.
    - **Select the type of installation required:**
        - NW Server Installation (choose this)
        - Client
        - Diskette Creation
        - Readme Files
    - **Choose product to install**
        - NW 4.11 (choose this)
        - NW 4.11 SFT III
    - **Type of install:**
        - Simple installation NW 4.11 (choose this)

- Custom
- Upgrade NW 3.1x to 4.x

- **Specify server name:** you can choose any name from 2 to 47 characters, using alphanumeric characters, hyphen or underscore. I named my server IDORU.
- At this point the server boot files get copied to the DOS partition. On the P166MMX with 32 MB RAM I am using this took about 3 minutes.
- After the copying of the files, NW tries to autodetect your hardware and install the right drivers for it automatically. If for some reason it cannot determine the correct driver for some of your hardware (e.g. a network card), it will ask you to choose the driver. If this happens, make sure you are well documented: you will need to know the IRQ and Base Address of your card, the manufacturer and type. If the drivers are not present on the CD, you will need a disc that should be provided by your manufacturer. You can also try to find the drivers on the Internet, which is usually the fastest solution.
- When all the correct drivers have been installed, you get a summary of the Server Drivers. At this point you can install additional drivers, for example if some piece of hardware was not autodetected.
- Now more files are copied to the server from the CD.
- **Is this the first NW4 Server?** The installation program asks if this server is the first NW 4.x server on your network. If it is, choose yes, otherwise choose no.
- **Timezone?** Choose the timezone you're in.
- **Organisation?** This is how your directory tree will be called. Choose a preferably short name. I named my directory tree ECOMM.
- **Admin Password?** You must type in a suitable password for the administrator account now, and re-enter it for verification. Don't write it down, as this is the password that allows administration access to the server. But don't forget it either. The normal rules for choosing safe passwords apply.
- At this point you get a summary of important information about your server. Write it down and store it in a safe place. This is what I got:
  - Directory tree name: ECOMM
  - Directory content: O=ECOMM
  - Admin name: CN=Admin.O=ECOMM
- **Insert Netware License Disc:** this disc was provided with your distribution of Novell, and it contains a file called SERVER.MLS which holds your license information.
- Now the "Main Copy" of the files takes places. In my case it only took a couple of minutes.
- After the Main Copy you get the chance to do some extra things like making diskettes. There's no need for that right now, just continue.

- You’ve reached the end of the installation! If you are lucky and don’t have any problems with drivers for hardware, you can finish it in half an hour. But if you have trouble like in my case, it could cost you a day or more. Murphy...

11.1.2 Installing Windows NT 4.0 over a LAN

Imagine this situation: you have a computer on your network, without CD-ROM drive, on which you want to install Windows NT. Obviously, you'd rather not start putting in a CD-ROM drive just for this installation. Well, you don't have to.

What you will need:

- a good backup of everything you still need from the local machine
- 120 MB of free space on your local (DOS) hard disk
- a bootable DOS system on that system (you can use a boot disk if you like). I've tried using a Win95 machine in DOS-mode, but this locked up the NT-installation program, apparently there was a problem with the long file names. So I booted up on a DOS disk, and that solved the problem.
- network access on your target PC - we will get the NT installation CD from the network, so you need a physical connection to it, plus the necessary software installed. I've tried it using a Novell Netware 4.11 server, and the DOS VLM-drivers to access the network. They fit on a 1.44 MB bootable DOS diskette, so you can create this disk to boot up any system, any time you want to install NT over the network.
- a NT installation CD (or a copy on a hard disk - make sure you have a license for it) accessible through the network.

When all the above requirements have been met, reboot the computer and logon to the network. Assuming *f:\temp* is the location of the NT installation CD, for installation on an Intel system change directories to *f:\temp\i386*.

Then type: "winnt /S:f:\temp\ /T:<local disk> /B", where <local disk> is any local hard disk with 120 MB free space. The /S option indicates where the installation program is to find the installation CD (this is a network path). The /B option allows "diskless" installation, to skip the use of the three installation diskettes otherwise needed.

In fact you can omit all parameters except the /B; the installation program will find any free space itself, and exit if there is not enough available. If you didn’t specify it on the command line, it will ask you to supply the source directory of the files (the f:\temp\ specified on the command line in the earlier example).

Once this is done, you enter the installation program which will guide you through the rest of the setup. First, the necessary files (120 MB) get copied to your local disk. This can take quite a while (on slower networks) because it's basically a bunch of small files. Using the DOS VLM client drivers, it takes one hour or more on a P166. After this, the installation continues, requiring you to reboot 2 times. Just follow the instructions of the program, and that's it!

There is a possibility to automate the installation process further by loading a file with all the choices pre-made, type "winnt /?" at the DOS-prompt (in the f:\temp\i386 directory in this case) to find out more about this option.

11.1.3 Configuring Netscape Enterprise Server 3.5.1

This is an overview of some of the configuration options available in Netscape Enterprise Server 3.5.1.

Conventions:

"Name" means click link or button Name

... means wait for next page

1) **Installing CGI file type**. This will allow files with extension .cgi, .bat and .exe to be executed no matter where they are in the server directory structure. The alternative is to add a CGI directory, so that any file located in that directory will be executable as a CGI program, regardless of its extension. This provides more safety but less flexibility as all users would have to put their CGI programs in that specific directory. It is possible to use both methods at the same time, but I prefer just to use the CGI file type.

Procedure:

Surf to your administration server.

Click on the server you would like to manage.

Click "Programs"

Click "CGI File Type"

Activate CGI as a file type? Click "Yes", "OK",...,"Save and apply"

2)**Activating Server Side Javascript.** This will allow the use of Server Side Javascript applications. These applications can be managed using the Server side Javascript Application Manager, a link to which you will find on the Server Side JavaScript configuration page after executing the procedure specified below. You can install up to 120 Javascript applications on one server.

Procedure:

Surf to your administration server.

Click on the server you would like to manage.

Click "Programs"

Click "Server Side Javascript"

Activate the Server Side Javascript application environment? Click "Yes"

Require administration server password for Server Side Javascript Application Manager? Click "Yes" "OK",...,"Save and apply"

3)**Configuration Styles.** Configuration Styles are an easy way to apply a set of options to specific

files or directories that your server maintains. You can specify the following options in a configuration style:

CGI file type
Character Set
Default Query Handler
Document Footer
Dynamic Configuration
Error Responses
Log preferences
Restrict Access
Server parsed HTML

See Netscape Enterprise Server Help for more information.

Procedure:

Surf to your administration server.
Click on the server you would like to manage.
Click "Configuration Styles"

4)**Configuring Document Preferences.** Here we can configure Index Filenames, Directory Indexing, the Server Home Page, Default MIME Type and the Parse Accept Language Header.

Index Filenames: default index.html and home.html. You can add some if you need to; you can also specify cgi scripts (if you have installed a CGI file type or a CGI directory).
Directory Indexing: this feature returns a list of the files and subdirectories of the requested URL if no document is specified in it and if there is no Index File available. Turn this off for security reasons. The server will generate an error instead of a directory listing.
Default MIME Type: If the server cannot determine the proper type of a document requested by a client, it returns the Default MIME Type in the section that identifies the document type.
Parse the accept language header: If you have copies of files on your server in multiple languages, a client using HTTP 1.1 can specify the language in which it would like to receive the requested file. The client sends header information describing the languages they accept. However, if you do not support multiple languages, you should turn this feature off as it slows down your server.

Procedure:

Surf to your administration server.
Click on the server you would like to manage.
Click "Content Management"
Click "Document Preferences"
Directory Indexing: Click "None"
The other features can be altered as you like.
Click "OK".

4)**Configuring URL Forwarding.** This feature allows redirection of URL requests to different URLs, for example if the location of (a group of) documents has changed. If you want to redirect a URL to a URL on another server, where the directory structure is the same, you can use the "URL Prefix" field. If the directory structure has changed as well, you can use the "Fixed URL" field. For instance, if you removed a user from your server, and you want to redirect all requests to any of his files to a file where you explain that the user no longer exists on this server, you would fill in (e.g.) http://some.server.name/olduser/ in the **URL prefix** field, and http://some.server.name/exists_no_more.html in the **Fixed URL** field.

How to get there:

Surf to your administration server.

Click on the server you would like to manage.

Click "Content Management"

Click "URL Forwarding"

5)**Setting up hardware virtual servers.** This is a way to have your server respond to multiple IP addresses without installing multiple software servers. I have tried this, but there seems to be some bug that stops my original server once I've added a hardware virtual server. I didn't get it to work.

6)**Setting up software virtual servers.** This is a way to host multiple sites on 1 server, using multiple hostnames, without the need for more than 1 IP address. This will only work with certain (newer) browsers. It does not work with Netscape Navigator 1.x. I have tested this feature with 2 browsers. It works with Netscape Communicator 4.04, but it does not work with Internet Explorer 2.0.

7)**Using Cache Control Directives.** Cache Control Directives allow to indicate to any caching proxy server which documents should be cached and which should not. The Proxy server must support HTTP 1.1. Possible levels of cache control are:

- Public
- Private
- No Cache
- No Store
- Must Revalidate
- Maximum Age

For more information on these levels see the Netscape Enterprise 3.5.1 Help.

Cache Control Directives can be set for the entire server or for specific files and directories. If your server hosts sensitive information it might be a good idea to restrict caching of that information.

For more information about HTTP 1.1, see the HTTP 1.1 specification, RFC 2068 at http://www.ietf.org/html.charters/http-charter.html

8)**Set up Netshare.** Netshare is a piece of software integrated in Enterprise Server that provides an Enterprise user with a personal home page.

9) **AutoCatalog.** This agent allows to catalog your web site: it automatically generates web pages that list and categorise the HTML documents on your server. The catalog can be accessed at http://some.servername.com/catalog. This feature does not seem to work; apparently a DLL (ns-httpd30.dll) is missing.

10) **Using encryption: SSL**. Secure Sockets Layer (SSL) is a protocol situated in the network layer, above TCP-IP, but below HTTP, FTP,... It uses a combined symmetric/public key encryption approach. Symmetric encryption uses one key for both encryption and decryption.

For more information on the encryption of SSL, see section 4.4.2.

In order to use SSL with NES, you need to obtain a certificate. How to obtain such a certificate is explained in-depth in the NES documentation, however, if you plan to use your NES for an Intranet, you can issue your own certificates using Netscape Certificate Server (NCS). Installing and configuring NCS goes beyond the scope of this document.

To obtain a certificate, go to Server Administration. Go to "Keys & Certificates". In order to request a certificate, we first have to generate a key pair, stored in a key-pair file. While generating the key pair, you have to specify an *alias,* which is a name associated with both a key-pair file and a certificate file. You use the alias to refer to these files when setting up SSL encryption on a server. The key-pair file contains an encrypted version of both the public and private keys used for SSL encryption. The file is used when you request and install a certificate. When you create the key, you will have to specify a password that will need when starting your secure server. Don't forget it.

To create a key-pair file: (from NES 3.51 On-line Help):

From the Windows NT command prompt:

1.Go to the <server_root>/bin/admin/admin/bin directory.
2.Run the sec-key.exe application. The key-pair file generation program appears.
3.When prompted, type an alias for the new key-pair file. You might choose an alias that matches your server (for example, web or mail). The alias cannot contain spaces, but it can use symbols that your operating system allows in filenames (such as hyphens and underscores). By default, the key-pair file is stored in the directory

C:/<server_root>/alias/<alias>-key.db where <alias> is the alias you typed. If you used the alias mail, your key-pair file would be C:/<server_root>/alias/mail-key.db.
4.A screen with a progress meter appears. Move your mouse in random motions at random speeds. These random movements are used to generate a random number for the unique key-pair file. 5.When prompted, type a password of eight characters or more for your key-pair file. The password must have at least one non-alphabetical character (a number or punctuation mark). Make sure you memorise this password. The security of your server is only as good as the security of the key-pair
file and its password. After you turn on SSL for a server (either the administration server or another
Netscape server), you must type the key-pair file password when you start the server.

Now you have to request your certificate: go to General Administration, "Keys & Certificates", "Request Certificate". Fill out the form and send the resulting "Certificate Request" to your Certification Authority (CA).

When you get your Certificate from your CA, go to General Administration, "Keys & Certificates", "Install Certificate". Choose "Certificate for this server" and paste the certificate text you got from your CA in the textbox. Be sure to select the correct alias and click OK.

After your certificate is installed, you can now turn on the encryption for your server. Usually, there is an insecure server running that handles with most of the requests. Only when it comes to exchanging sensitive information (Credit Card numbers, etc.), the secure server comes into action. So what I did is the following: I created a new server with the same hostname, but secure, i.e. with port number 443 (default for https). This means of course that you will have to re-do all the configuration changes you have made for the other server on the new server. But it has the advantage that the insecure and the secure part of your server are separated and easily accessible. The insecure server answers to http://some.server.name/, and the secure server answers to http**s**://some.server.name/.

11) **Setting access restrictions for your server.** The default setting for access restriction on NES allows any user you've added to the user database to create any directory under the document root. This is not desirable in most cases. So we want to restrict access to our server.
The best procedure is to start by denying "anyone" (all users, including the ones without authentication), from "anyplace", "all" rights. Then we allow "anyone" from "anyplace" the "r-x-li" (read, execute, list and index) rights. Then we allow "owner" (the owner of any file on the server) from "anyplace" "all" rights. This set of rules will allow anyone on your network to read and execute files/directories on your server, but only the owners of the files/directories will be able to change them, delete them or add new files/directories. Of course, you can

adapt these rules to your own needs, but keep in mind that it is always safest to start by denying all rights to anyone, and then gradually allowing certain rights to certain (groups of) people.

Procedure:

Surf to your administration server.

Click on the server you would like to manage.

Click "Server Preferences"

Click "Restrict Access"

A. Pick a resource: Editing: Choose "The entire Server"

Click "Edit Access Control"

Click "New Line" (this line denies all rights to anyone from anyplace)

Repeat Click "New Line" (alter this line to what you like) until you're ready

Click "Submit" when you have added all the lines you want.

...

Click "Save and apply"

11) **Using Perl scripts with NES.** To be able to use Perl scripts with NES, we need to install Shell CGI Programs. Shell CGI is a server configuration that lets you run CGI applications using the file associations set in Windows NT. As with the Installing of CGI types (see 1), we have 2 options. Either we specify a shell CGI directory, or we configure the server to associate specific file extensions with shell CGI by editing MIME types from the Server Manager. We choose the last approach, to allow more flexibility for our users. But even if you want to allow your users to execute shell CGI programs anywhere, you'll have to create a shell CGI directory first, because this will activate shell CGI programs for your server. This problem is not documented in NES.

Procedure:

Surf to your administration server.

Click on the server you would like to manage.

Click "Programs"

Click "ShellCGI Directory"

First, we will add a shell CGI directory. Fill in an URL prefix (e.g. cgi-shell) and the physical Shell CGI directory on your server's hard disc. The directory doesn't have to exist on the hard disc if you don't intend using it.

Click "OK"

...

Click "Save and Apply"

Now, we will create the perl MIME type.

Click "Server Preferences"

Click "MIME type"

There is an entry for the .pl extension by default. However, to associate .pl files with the installed Perl for Windows NT, we need to edit it. So look up the entry in the table and Click "Edit".
Choose "type" from the drop down box, and set the content type to "magnus-internal/shellcgi". The file suffix field remains unchanged, "pl".
Click "Change MIME type"
...
Click "Save and apply"

12) **Deleting users with Netshare.** To delete a user and his/her Netshare, you start by deleting the user in General Administration, Users and Groups, Manage Users. Then delete the Netshare directory of the user on the server, using Explorer (on a Windows system) or the command prompt. Now you have to go to Server Administration, Web Publishing, Index and Update Properties. Re-Index the Netshare directory, so that Netshare will notice that the Netshare directory of our removed user is gone.
Procedure:
Surf to your administration server.
Click "Users and Groups"
Click "Manage Users"
Find the user that you want to delete
Click "Delete User"
Now remove the directory using Explorer or the Command Prompt
Surf to your administration server.
Click on the server you would like to manage.
Click "Web Publishing"
Click "Index and Update Properties"
Click "View"
...
Select your Netshare root directory
Click "Index"
...
Remove the checkmark at "Set document owner to"
Click "Index"
Done!

13) **Setting document ownership of many documents.** When you want to set the ownership of a lot of documents, you have to re-index the documents, filling out the "Set owner documents to" dialog box. This seems to be the only way to take ownership of directories, since the View|Properties feature in the Netscape Web Publisher doesn't seem to work for directories. But of course when a user creates a new directory in his/her account

using the Netscape Web Publisher, he or she takes ownership immediately. Though most of the times you have to do a View|Reload Window before you see the ownership.

Procedure:

Surf to your administration server.
Click on the server you would like to manage.
Click "Web Publishing"
Click "Index and Update Properties"
Click "View"
...
Select the directory you want to re-index
Click "Index"
...
Fill in the "Set document owner to" field
Click "Index"

14) **Using Java Servlets with NES.** This feature has been built into NES 3.51. The Netscape Virtual Machine (VM), under which the servlets run, supports JDK 1.1. All you have to do to get the Servlets to work, is to set the servlet directory and turn Java on for the server.

Once you have activated the Java Servlets, you can access them through http://<server-name>/servlet/<servlet-name>. If you set the directory of the Java servlets to <ns-home>/plugins/java/servlets (<ns-home> is typically "/Netscape/SuiteSpot"), then you will be able to try the example supplied with NES, BrowserDataServlet. Otherwise, you will have to copy the example into your servlet directory first. I added one more setting to the obj.conf file (<ns-home>/https-<host>/config/obj.conf, where <host> is the name of your server as listed in Netscape Server Administration). This is what you will find at the bottom of the document:

```
<Object name="servlet">
Service fn="java-run" class="sun/servlet/netscape/NSRunner" vpath="/servlet"
</Object>
```

I added the field *initfile* (line 2 and 3 of the following code should be on one line):

```
<Object name="servlet">
Service fn="java-run" class="sun/servlet/netscape/NSRunner" vpath="/servlet"
        initfile="<ns-home>/https-<host>/config/servlets.properties"
</Object>
```

Here you will have to exchange <ns-home> and <host> for the appropriate directories. Check the documentation of the JSDK for information on the content of the servlets.properties file. You can use servlets without this file.

If you make this change in the obj.conf file, you have to make the server know you have done it manually. Do the following:

1) Surf to your administration server.
2) Select the server you wish to manage.
3) Select the Apply button in the upper right hand corner.
4) Click on the Load Configuration Changes button.

Since version 1.2 of the JDK (currently (March 98) still in beta stadium), the JSDK is incorporated in the JDK. Before that, the JSDK (v1.0.1) was available as a separate package. This package can still be found at http://jeeves.javasoft.com. But other than for the documentation and extra examples, there is no need for the JSDK to use Servlets with NES 3.5.1. If you want to write big applications, it is worthwhile to look into one of the free Servlet plugins like JRun, by Live Software. They greatly improve the rather limited implementation of Servlets of NES 3.5.1.

Procedure:
Surf to your administration server.
Click on the server you would like to manage.
Click "Programs"
Click "Java"
Check "Yes" to activate the Java interpreter, and supply the directories of the Java server side applets and Java servlets you want to use.
Click "OK"
...
Click "Save & Apply"

15) **Using NES to search documents.** NES includes a search engine that allows to search all documents on the server, that is, the ones that are indexed. By default, your whole document root directory is indexed, and can be searched. If you create a new URL on your server which points to a directory that is not located under the document root, you will have to make a new Collection. This will index the documents you specify (e.g. *.html under e:/javatut), making them searchable from the web.
You cannot make a Collection of a directory that is not a document directory. So before making a new Collection of some new directory, be sure to add it to the Document Directories list.
Remark: to search the new collection, you will have to select it in the "Search In" drop down box, that can be found on the default search page at *http://<server-name>/search*.

Also make sure that you update the index of the searchable pages on a regular basis; this can be done automatically using the "Schedule Collection Maintenance" page under "Agents & Search" in the Server Manager.

Procedure:

Surf to your administration server.

Click on the server you would like to manage.

Click "Agents & Search"

Click "New collection"

Fill in the fields "Directory to index", "Documents matching", "Include subdirectories Yes/No", "Collection Name", "Collection Label", "Description", "Collection Contains", "Extract Metatags" and "Documents are in".

Click "OK"

...

Click "Save and Apply"

11.1.4 JBuilder 1.1

JBuilder is Borland's answer to Symantec's Café and Microsoft's Visual J++. With a GUI that will look a bit familiar - but is still quite different - if you've ever used Delphi, Borland has managed to create an Integrated Development Environment (IDE) that allows genuine Rapid Application Development (RAD) using Java: you can build a full-featured graphical interface by drag-and-drop.

JBuilder includes an excellent class browser, neatly integrated with the new "AppBrowser".

This AppBrowser is a whole new approach to the efficient writing of code: it lets you explore, edit, design and debug, all in one unified window. The AppBrowser usually contains three panes: the Navigation pane on the upper left, the Structure pane on the bottom left and the Content pane on the right.

The Content pane has a set of tabs at the bottom, allowing you to select a viewer or editor for the current file. E.g., when the current file is a .java file, there are three viewers available: the Source Code Editor, the UI Designer (if this class has a UI) that allows the visual construction of the UI of this class and the Documentation viewer, in which Borland created an enhanced version of the JavaDoc standard by Sun. It automatically generates HTML documentation pages from your java source code and comments.

The Structure pane has a set of tabs allowing you to select what *kind* of browser it is. The Project Browser is one of the most important, as it allows you to browse through the files of your project, but also the structural elements of the files in your project, e.g. a certain function in one of your classes, or the line of code where a particular button is defined. This allows very speedy browsing of your project, without having to scroll through source files manually. The Structure pane also contains the Debugger and the Class Hierarchy Browser.

JBuilder works with classes and packages, obviously, as this is a Java package. Your project is a package (it is possible to have more than one project open, this was an annoying limitation of the older Delphi versions), that consists of several classes.

One thing is annoying when visually designing your forms: it is impossible to copy components - like buttons etc. - visually from one class to another using Windows' copy and paste. Borland has promised that this will be possible in the new version 2, which was announced on 23rd March to be available in "early spring". We'll see. Something else that will be fixed in version 2, is the support for other JDK's. In version 1.x, you have to use the classes Borland provided with the product - there is no way to compile against a newer version of the JDK. In fact, Borland warns explicitly in the documentation that you should not replace the classes.zip file with a newer version from Sun.

There is another seriously annoying thing about JBuilder. It's the Online Help. Borland has a tradition of having Online Help files that are not very extensive (the old Turbo Pascal packages and Delphi have this problem). This is however not the case now, maybe because the help is actually the Java specification and class definitions/descriptions from Sun. The problem is the interface. Borland has chosen not to use the standard windows help files, but has written its own Java help browser. And that has some serious flaws: once you are in it, Alt-Tab will not switch you back to the IDE, and this help browser does not show up in the list of open applications you get when you press Alt in the IDE. Another problem is that when you select a topic in the list by typing the first few letters, half of the times the corresponding information does not show up in the browser window when you press enter. You have to click on another item in the list, so that the full name shows up in the editable text field, and then click the item you want so that its full name shows in the text field and press enter or double-click. Seriously maddening - I hope this will be fixed in version 2.

Looking at performance, I can say the following. I installed JBuilder on a P166MMX with 32MB of RAM running NT 4.0 Workstation. Borland recommends more RAM, and they are right. It works, but quite slowly by times, e.g. when you switch to a different viewer. Compiling the code goes very fast though - a lot faster than C and maybe even a bit faster than Delphi. Executing the Java bytecode was not as fast though, but I suppose if you compiled the bytecode to platform dependant machine code, this performance issue would not be a problem anymore. Executing the application outside the IDE might be a good solution as well.

Being new to Java, I decided to do the tutorial provided with JBuilder, which consists of the creation of a basic text editor. I succeeded easily, although the description of the different steps was a bit brief sometimes. Some parts of the blocks of code were not explained at all. There were also quite some errors in the code - it is built up in steps, but some of the lines were introduced a few steps too early, resulting in compiler errors. But if you just comment out the offending lines, it compiles, and a few steps later you will see when to uncomment it. In the end I had a working text editor.

I then decided to go a step further, just to get to know JBuilder better and to learn some Java. I found a Java encryption class on the Internet (http://munkora.cs.mu.oz.au/~mkwan/), called Information Concealment Engine (ICE), by Matthew Kwan. It is freeware, and he describes it as follows:

> "It is a 64-bit private key block cipher, in the tradition of DES. However, unlike DES, it was designed to be secure against differential and linear cryptanalysis, and has no key complementation weaknesses or weak keys. In addition, its key size can be any multiple of 64 bits, whereas the DES key is limited to 56 bits."

I decided to include the ICE class in the text editor I had written, to allow encrypted storage of text files. The integration went seamlessly; all I had to do was add a new file to the project (the icepack.java file), and then add one line at the top of this class file ("package textedit;"), to allow referencing to it from the other files in my project. After some work, I have now an application that allows the encrypted storage and reading of text files, with several encryption levels.

11.2 Source Code

The source code can be found on the disk accompanying this report. All the source code has been formatted in such a way that it will print without unwanted linewraps when you copy it into a MS Word document with font Arial and fontsize set to 8.

11.3 General.css

This Cascading Style Sheets file is used throughout the site to format forms. The file can be found on the disk accompanying this report.

11.4 Used Abbreviations

ACL: Access Control List

DECT: Department of Engineering and Computer Technology (at the UCE)

FTP: File Transfer Protocol

HTTP: HyperText Transport Protocol

IE: Microsoft Internet Explorer

JDK: Java Development Kit

JSDK: Java Servlet Development Kit

KIHO: Katholieke Industriële Hogeschool Oost-Vlaanderen

LAN: Local Area Network

MS: Microsoft Corporation

NES: Netscape Enterprise Server

Netscape: Netscape Communications Corporation

NT: New Technology, in this book short for Windows NT

OS: Operation System

SSL: Secure Sockets Layer

SUN: Sun Inc.

UCE: University of Central England

Y2K: Year 2000

12. References

The Apache Site (http://www.apache.org)

The eMarketer Inc. Site (http://www.emarketer.com)

The Inprise Corporation Site - formerly the Borland Inc. Site (http://www.inprise.com)

The Internet Engineering Task Force (http://www.ietf.org)

The Javasoft Site (http://www.javasoft.com)

The Live Software Inc. Site (http://www.livesoftware.com)

The Lotus Development Corporation Site (http://www.lotus.com)

The Microsoft Corporation Site (http://www.microsoft.com)

The NCSA Site (http://www.ncsa.uiuc.edu)

The Netscape Communications Corporation Site (http://www.netscape.com)

The Novell Inc. Site (http://www.novell.com)

The Novonyx Inc. Site (http://www.novonyx.com)

The original SSL proposal by Netscape (http://home.netscape.com/newsref/std/SSL.html)

The Sun Inc. Java Site (http://java.sun.com)

The World Wide Web Security FAQ (http://www.w3c.org/Security/faq/www-security-faq.html)

The World Wide Web Consortium Site (http://www.w3c.org)