J2EE Architecture and Patterns in Enterprise Systems

Zhiguo Guo

University of Tampere
Department of Computer and Information Science
Master’s Thesis
May 2004
The purpose of this thesis is to illustrate an enterprise project named HUVA, in which Java Platform 2, Enterprise Edition (J2EE) architecture and patterns are applied. This study took a constructive research approach: It starts from the application domain including the initial plan and business model, then moves to the system architecture covering the architecturally significant requirements and the system structure compliant with J2EE architecture, finally ends at the application design elaborated by applying the J2EE patterns at a high level. As a result the architectural and application design satisfied the project needs and requirements; and the HUVA software has been implemented in practice according to the design.

Keywords: J2EE, architecture, pattern, enterprise system.
Acknowledgements

I would like to thank Professor Roope Raisamo and Professor Isto Aho from the University of Tampere for being my encouraging, patient and responsible supervisors during the master’s thesis project.

I would like to thank Mr. Mika Miettinen from Solteq Oyj and Mr. Timo Westkämper for supporting the technical aspects of this thesis.

Finally, I would like to use this opportunity to express huge thanks to my dear wife, Chongliang Tang for always being the invaluable person behind my success.

Tampere - Finland, June 1st, 2004

Zhiguo Guo
Contents

1. Introduction............................................................................................................. 1

2. Application domain ............................................................................................... 3
   2.1. Project background ....................................................................................... 3
   2.2. Development needs ...................................................................................... 3
   2.3. Customer requirements .............................................................................. 4
   2.4. Development environment .......................................................................... 9
       2.4.1. Technology ......................................................................................... 9
       2.4.2. Topology ........................................................................................... 10

3. System architecture .............................................................................................. 13
   3.1. Introduction to software architecture ............................................................. 13
   3.2. Architectural design considerations ............................................................... 14
       3.2.1. What is architectural design? ........................................................ 14
       3.2.2. Architecturally significant requirements ..................................... 14
       3.2.3. Top-down design approach ........................................................... 15
       3.2.4. Classification of layers and tiers ................................................... 15
       3.2.5. Establishment of services ............................................................... 17
   3.3. J2EE overview .............................................................................................. 17
       3.3.1. What is J2EE? ................................................................................... 17
       3.3.2. What is J2EE technology? ............................................................... 18
   3.4. HUVA system architecture .......................................................................... 20
       3.4.1. HUVA architecturally significant requirements ......................... 21
       3.4.2. HUVA layers, tiers, and services .................................................. 24
       3.4.3. HUVA system structure ................................................................. 27

4. Application design patterns ................................................................................ 30
   4.1. Introduction to pattern ............................................................................... 30
   4.2. J2EE patterns overview .............................................................................. 31
       4.2.1. Presentation tier patterns ............................................................... 31
       4.2.2. Business tier patterns ...................................................................... 32
       4.2.3. Integration tier patterns ................................................................... 32
       4.2.4. J2EE patterns relationships ............................................................ 33
   4.3. HUVA presentation tier patterns .................................................................. 35
       4.3.1. Presentation tier design considerations ...................................... 35
       4.3.2. Services to worker ......................................................................... 39
   4.4. HUVA business Tier patterns ....................................................................... 44
       4.4.1. Business tier design considerations ......................................... 44
       4.4.2. Business delegate and session façade ...................................... 45
       4.4.3. Service locator .............................................................................. 47
4.4.4. Business object and transfer object ........................................ 48
4.5. HUVA integration tier patterns .................................................. 49
  4.5.1. Integration tier design considerations ................................. 49
  4.5.2. Data access object ......................................................... 51
  4.5.3. Domain store .................................................................. 54
5. System review .......................................................................... 55
6. Summary .................................................................................. 60

References ....................................................................................... 61
1. Introduction

With use of the Internet, web browser-enabled software applications are developed rapidly. Nowadays enterprises are paying more attention than ever on developing their business on the Web. The demand for rapidly delivering high quality web enabled software applications to the end users is high. The enterprise software developers are making efforts to develop the enterprise software applications that not only satisfy the business needs but also achieve the high quality within a short development process. However, the enterprise software development is normally complicated. For example, factors like the different enterprise environment, the critical business demand, the lack of knowledge and experience from the developers and so on can make the enterprise software development become rather hard work. Under such circumstances, something is needed to simplify the enterprise software development and also enable the sharing of the knowledge and experiences among the different developers.

J2EE stands for Java 2 Platform, Enterprise Edition, designed by Sun Microsystems for developing distributed enterprise software applications. As agreed together with industry partners such as IBM, J2EE has become a standard industrial architecture framework aimed to simplify enterprise application development. J2EE defines a standard architecture composed of an application model, a platform for hosting applications, a Compatibility Test Suite (CTS) and a reference implementation. In general, J2EE architecture is a tiered architecture with the primary concerns of application components, containers, resource manager drivers, and databases in its running environment.

To capture design experience, in software design and construction, in a form that people can use effectively was the idea of design patterns given by the book *Design Patterns: Elements of Reusable Object Oriented Software* [Gamma et al., 1994]. Before this, patterns were first described by Christopher Alexander [Alexander, 1977], and applied to the construction of buildings. In the book *A Pattern Language*, Alexander writes:

> Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of solution to that problem, in such a way that you can use this solution a million time over, without ever doing it the same way twice.

Along with the idea of patterns, J2EE patterns are a set of design patterns the software developers can use in the context of designing Java 2 Platform, Enterprise Edition (J2EE) applications. In other words, the J2EE patterns are a
collection of J2EE-based solutions to common problems, and reflect the collective expertise and experience of Java technology architects over the past years. Each pattern hovers somewhere between a design pattern and an architectural pattern [Alur et al., 2003]. Through the use of the tiered approach, J2EE patterns are clarified according to functionality into three logical architectural tiers: presentation, business and integration tier. Each of J2EE patterns is rarely used along; instead they are used in combination to solve common problems.

In the thesis, an enterprise project named HUVA in the field of customer service is designed by applying the J2EE architecture and patterns. Some issues related to software architecture and design patterns within J2EE are addressed. The main idea is to show illustrate how J2EE architecture and patterns can be used to build the enterprise systems to achieve the architecturally significant requirements.

After the introduction, the application domain is presented in Chapter 2. More specifically, some background information about the HUVA project is first introduced. Afterwards the HUVA business model is presented. At the end, the specific technologies and development environment provided by the company and available for the project are explained. The following Chapter 3 and 4 are main parts of this thesis. In Chapter 3, the architecturally significant requirements of HUVA are discussed; and the HUVA architecture is designed, with the respects to the specific application environment, to match the architecturally significant requirements. In Chapter 4, along with the architecture designed in the chapter 3, J2EE patterns are used to provide the proven solutions to the recurring problems in the context of HUVA implementation that reflects the HUVA J2EE architecture. The Chapter 5 reviews the HUVA project, with a focus on the design work done in the two previous chapters. In Chapter 6, this whole thesis project is summarized.
2. Application domain

This chapter introduces the HUVA enterprise project and its domain model that is to be implemented during the project. At first, some background information related the project is presented. Afterwards the initial development needs and customer requirements are followed. At the end, the specific technologies and development environment provided by the company are explained.

2.1. Project background

Solteq car division, as a corporate branch under Solteq Oyj, is specialized in providing integrated solutions for car dealers in Finland. The Enterprise Resource Planning (ERP) system being offered to its customers, the car dealers, is called Solteq CD, which covers the operative functions of car dealers and supports multi-outlets and multi-warehouses environment. More specifically, The CD system handles car sales, spare part and repair shop functions, finances, bank connections and customer information.

As the system was becoming widely used by its customers, Solteq car division found that car service reservation, as the part of the service function, was missing in its CD system. The car division also noticed that the car service reservations at its customer premises were normally handled with pen and paper in practice. An idea to replace this paper work with electronic forms was brought up inside the company. At the same time, a significant customer of the Solteq car division noticed the same situation in the process of making car service reservations, while this customer was in growth of the car business. After getting to know the idea from the Solteq car division, this customer wanted to take part in the planning and development of this system by providing the real business environment to the developing team of Solteq. Therefore, there had been a decision to develop a system to replace the paper work used in the car service reservation process with electronic forms. At a high level, this system is an extension of the enterprise legacy system. Soon a project named HUVA was formally set up and that customer was in commitment to the project.

2.2. Development needs

Conceptually, HUVA project was intended to replace the manual paper workflow for making car service reservations and aimed to help the car service businesses improve the service quality, manageability and productivity. The final outcome of the project should a software system that handles the car services reservation process at car repairing shops and should be used by its
end users at ease and with great flexibility. At this point, a software application with web browser-enabled user interface was appreciated by most of the decision makers at the company. In addition to satisfy the participated customer, the software was initially planned to target, in scope to domestic customers, auto-repairing shops in Finland.

2.3. Customer requirements
Functionally, the HUVA project had to implement an online business model of car service booking. The business model consists of service booking, booking info related search and workload management.

Basically there are two types of users involved in this software. One is the car service agent who makes the car service reservation requested by the customer; the other is the administrator who manages workforce, arranges work shifts and so on.

The following figure 1 and 2 are the high-level use case.

---

**Figure 1. Car service agent use case.**

Figure 1 shows the car service agent browsing, adding and updating the service booking. Each time when a booking is added or updated, a confirmation is included.
The administrator takes care of the workforce management. It includes editing the job shift info and updating the shift arrangements.

Like mentioned earlier on page 3, the overall business environment is that HUVA project is meant to be a service extension of the integrated Solteq CD system. The figure 3 provides a picture about the HUVA relation to the CD system.
The Business model that HUVA presents is a service reservation process in car service shop. Figure 4 describes the reservation making process in general with respect to the organizational functions of a car service shop.
When a car service request arrives, the car service agent will first collect the information about the car, customer and payer. After this information registration, it is time to decide what to do with the customer’s car. Initially the customer will claim what to do with the car. At the same time, the service agent will give advice and supplementary comments, according to the customer’s
initial request. In practice, not every customer knows what to do with his/her car when requiring the services. At this point, the care service shop normally provides a pre-checking service before defining the real service tasks. The pre-checking service is performed by professional car mechanists. Through this pre-checking, right services for the car can be defined. The customer can have options to choose the type of service between time log and position required. The time log required means the service is carried out within specific time periods, while the position required means the service is be carried out by certain service team or serviceman. At this point, the workforce information should be available for the car service agent to book the working team and time. Once the reservation is made, the information is available for browsing by the car service agents and other administrators involved.

As we can see, it is not uncommon to implement such a business model. However, besides the functional requirements, there are still some specific concerns that should be well considered.

Since the day that the HUVA was setup, a practical investigation on modeling the process of car reservation making has been undertaken by the developing team. From the practice, the developing team discovered that the way of handling the car service reservation with paper and pen has been in use for a long period; and operators or the car service agents at the customer’s premises are quite proficient with the paper facilities. To convince the customers that there will be a better way to handle such business process and increase the productivity, the HUVA software must be:

- efficient: compared to the paper forms and drawings, the software must provide a faster and more efficient means to record the car service reservation.

- graphically visualized: In practice, the car service reservations are being made daily. The software is used regularly and frequently/intensively by the users. For large scaled car service shops, there will be a large number of reservations every day at different service branches. The car service agent needs to grasp not only the overall workload situation at glance, but also the detailed workload situation for each group, team and employee. Normal textual reservation records slow down the car service agent’s workflow. How to visualize the large number of reservation records directly influences the work efficiency mentioned above.

- available: Some customer might have more than one car service shops, any car service agent should be able to have the information and make the reservation across different shops at almost any time.
• flexible: The software should be kept simple and flexible in use. The whole system should be kept adaptable, as the business keeps developing.

• reusable: the software is aimed to improve the efficiency and to reduce the workload. Data being stored from the reservation making process should be available for use elsewhere on other purposes. The software system itself should be well designed and can enable reuse for other development purposes.

2.4. Development environment
As being an advanced partner of IBM Corporation, Solteq Oyj setup its java environment backed with the IBM hardware for the development of the enterprise applications. Based on the J2EE platform, the technologies such as Servlet/JSP technology, Enterprise Java Bean technology, Java Database Connectivity (JDBC), Java Naming and Directory Interface (JNDI), and Java Transaction Service (JTS) are together available for use to develop the web browser-enabled enterprise applications.

2.4.1. Technology
The Java technology is defined, at Sun Microsystems, as a portfolio of products that are based on the power of networks [Sun Java Technology, 2004]; and the idea that the same software should run on many different kinds of systems and devices. It means not only a programming language, but also a selection of specialized platforms. It standardizes the development and deployment of the kind of secure, portable, reliable, and scalable applications demanded by the distributed network environment. Any Java application can easily be delivered over the Internet, or any network, without operating system or hardware platform compatibility issues.

The Java platform is a software-only platform that runs on top of other hardware-based platforms. In general, Java platforms can be mainly seen as three specialized platform editions:

• Java 2 Platform, Standard Edition (J2SE), provides an environment for Core Java and Desktop Java applications development, and is the basis for Java 2 Platform, Enterprise Edition (J2EE) and Java Web Services technologies. It has the compiler, tools, runtimes, and Java APIs that let the software developers write, test, deploy, and run applets and applications.

• Java 2 Platform, Enterprise Edition (J2EE), defines the standard for developing component-based multi-tier enterprise applications. It is
based on J2SE and provides additional services, tools, and APIs to support simplified enterprise applications development.

- **Java 2 Platform, Micro Edition (J2ME)**, is a set of technologies and specifications targeted at consumer and embedded devices, such as mobile phones, personal digital assistants (PDA's), printers, and TV set-top boxes.

Servlets and JavaServer Pages (JSPs) are components used to handle HTTP requests from web clients. In general, the presentation logic of a web application is processed by them. The Servlets and JSPs are deployed, managed and executed on a J2EE server and are generally called “Web components” [J2EE Overview, 2003].

Enterprise JavaBean (EJB) components are also deployed, managed and executed on a J2EE server. Their environment must support transactions. Enterprise beans usually contain the business logic for a J2EE application; they interact with the databases. The EJB architecture described in the EJB specification defines the contract enabling Java code written by multiple vendors to interoperate in a distributed programming environment [EJB Specification, 2004]. The reusable code components are referred to as Enterprise JavaBeans. EJBs provide the scalability and reliability aspects inherent to industrial strength server applications. To support the scalable, secure, client-server applications, the Java technologies like JDBC, JNDI, JTS, Servlet/JSP, RMI and JMS are incorporated by EJB.

### 2.4.2. Topology

The total environment for the development of enterprise applications at Solteq Oyj is shown at Figure 5 below. The execution environment consists of a HTTP Server, an IBM WebSphere Application Server and data storage facilities such as DB2 and MQ. The WebSphere Application Server (standard or advanced) provides the execution environment. A configuration program (the Administrative Console) is used to configure the Application Server with Nodes and servers, JDBC drivers, Data sources, Web applications with Servlets and JSPs and the JSP compiler. For each Web application, directories are set up where HTML files, JSPs, Servlets and JavaBeans must reside. The WebSphere Server provides a file server utility that can serve HTML and JSPs from a Web application directory instead of using the HTTP Server directories.
The WebSphere Application Development Studio provides a complete development environment for Web applications consisting of HTML, JSPs, Servlets, and JavaBeans and EJS container holding the EJBs. The Studio workbench contains projects with folders that contain files. The folders can be organized in any desired structure. The projects are stored in a directory structure on a local or LAN drive. The Studio itself holds a testing environment for the entire enterprise applications. The studio also provides three wizards that can be used to generate HTML, JSP and Servlet code for interaction with a
relational database or a JavaBean. Connected to WebSphere Application Studio is the source management server, which is controlling the source version during the development process.
3. System architecture

The main purpose of this chapter are to discuss the architecturally significant requirements (ASRs) of the HUVA system and to present the HUVA architecture, in addition to a brief introduction to architecture and J2EE. In the arrangement of the content, Software Architecture is first introduced. After the introduction, some architectural design considerations are discussed, for instances the issues of top-down approach, classification of layers and tiers, and establishment of services. The J2EE, as an industrial standard for the development and deployment of enterprise applications, is presented with a focus on its technology and architecture aspects. For the rest of this chapter, the HUVA architecturally significant requirements and architecture are presented and explained at its architectural level.

3.1. Introduction to software architecture

According to IEEE Standard 1471-2000 [IEEE, 2000], software architecture is the fundamental organization of a system embodied in its components, their relationships to each other and to the environment, and the principles guiding its design and evolution. Another commonly accepted definition [Boehm et al., 1995] is that a software architecture comprises:

- A collection of software and system components, connections, and constraints.
- A collection of system stakeholders' need statements.
- A rationale which demonstrates that the components, connections, and constraints define a system that, if implemented, would satisfy the collection of system stakeholders' need statements.

Actually, there is no standard, universally accepted definition of the term, software architecture. Numerous definitions can be found around the world through different channels.

In general, software architecture is concerned with different software parts and their relationships, structure and behaviour, static and dynamic structure, different viewpoints and levels, solutions and their rationale, and rules and principles on system development. More simply, it is about how to design software components and make them work together, and what rules to be obeyed when changes of software system are in need as the software system increases. It is just for exactly the same reasons given by Dijkstra and Parnas: Structure is important, and getting the structure right carries benefits [SEI, 2002].
3.2. Architectural design considerations

3.2.1. What is architectural design?
An interesting question here before going any further with this paper is how architecture is different from design. Paul Clements pointed out that architecture is design, but not all design is architecture [Clements et al., 2002]. Architecture design refers to the high-level structural design of the software application. It establishes a system skeleton at some high level and the constraints on downstream activities; and those activities must turn out artifacts (e.g. the detailed design document and its implantation) that are compliant with the high level design. Note that the architectural design does not define any implementation.

The issues brought up while designing a software architecture are communication between subsystems, data access and persistence, assignment of functionality to components, physical distribution, scaling and performance, future adaptability, reusability and so on. Architects deal with these issues by utilizing the architectural concepts of abstraction, decomposition, composition, style, and even aesthetics. At the informal note, the architect’s personal experience from the practice might influence the final decisions to cope with the above issues during the architectural design process.

3.2.2. Architecturally significant requirements
The Rational Unified Process® (RUP®) [Eeles, 2001] defines a requirement as follows:

A requirement describes a condition or capability to which a system must conform; either derived directly from user needs, or stated in a contract, standard, specification, or other formally imposed document.

A software system may have many requirements from various perspectives. Before the architectural design, these requirements are normally identified and grouped or classified. In real life, the most common way to classify the requirements is to keep functional (what to do) and non-functional/quality (how well it does) requirements separated. No matter whatever and however the requirements are classified, the architecturally significant requirements are referred to the ones in which the architect is most interested, and dominate the overall architecture.

In modern software development, architecture makes sense through two means. The first is that architecture guides the software development process. Besides satisfying functional requirements, the essential quality attributes like
reusability, security, performance, scalability, price, maintainability and so on are achieved through architectural solutions. It means that new architectures can be invented and the existing ones can be evolved. The second is that a software application is built using the structures and mechanisms provided by an existing architecture. The essential design problem is how to map the requirements to the architecture rather than to the programming language. The architecture takes the role of the implementation platform. By this means, product-line approaches, which provide a based architecture at a high abstraction level for the similar product families, are becoming common.

3.2.3. Top-down design approach

In software engineering, a common design approach is top-down approach. The top-down design is a software design technique, which aims to describe functionality at a very high level, and then partition it hierarchically, as the design develops, into more detailed sub levels until the detail is sufficient to implement the whole software. In other words, top-down design captures the overall design information in one centralized location or super level. By doing so, it is easier to make some significant design changes during the design process as all sub levels are listening or linking to the super level.

Bottom-up approach, as an approach opposite to top-down, attempts to drive the system from the source code. In practice, it is sometimes true that the top-down approaches become expensive to develop the domain models and it is difficult to develop models that are sufficiently general to be applied to multiple systems, as some enterprises have not set up a stable development environment and process. The development of the domain models is done case by case, not scientifically and systematically. Under such circumstances, making up a software architecture by gathering all the working parts together seems efficient and cost-effective, however it damages the software architecture qualities. These enterprises will achieve the worse result in long term. The top-down requires some expensive costs at the beginning, but it benefits the enterprises by building the product-line architecture in long term.

3.2.4. Classification of layers and tiers

Layer and tier are nowadays frequently mentioned by architects, engineers and developers when building enterprise systems. It is important to find out what they are in term of architecture and how to classify them.

Layers in term of architecture means hierarchical units in a software division. The units are layers; each layer represents a virtual machine, which is a collection of software that together provides a cohesive set of services that other software can utilize without knowing how those services are
implemented [Bachmann et al., 2000]. One of the fundamental properties of layered system is that the virtual machines are created to interact with each other according to a strict ordering relation like “A layered system is organized hierarchically, each layer provides service to the layer above it and serving as a client to the layer below.” [Shaw and Garlan, 1994].

Comparing to layer, tier is a whole functional entity and resemble layers, it may also provide different services whereas a layer is also a functional entity but of a much smaller context, limited to providing a building block for some service. Each tier normally resides on a separate machine.

Figure 6 below shows that enterprise applications can normally be divided into different tiers as the building blocks. Each tier can be physically distributed on different machines. To make each tier itself working, different layers interact with each other. Within a software application, the similar layers can be found in each tier.

As multi-tiered and multi-layered application design is commonly accepted by the enterprises and their engineers, the thing that concerns the software engineers is how to assign the functionalities/responsibilities to the defined tiers and how to utilize the services provided. The tiers in an enterprise application can be broken down into client, presentation, business, integration and resource tier [Alur et al., 2001], when considering a tier as a logic partition.
of the separation of concerns in the system. Each tier takes the different responsibilities, which hold the whole software system.

Client tier: takes the responsibility to interact with human beings. The final type of the clients can be different e.g. mobile device, web-browser and command terminal.

Presentation tier: is responsible for presenting the business information generated by the business tier. According to the type of client in the client tier, the presentation type can be different. However, the business data that this tier attempts to present is always the same.

Business tier: models the business process, and take the responsibilities to process the logic and deal with the data. It provides the business services required by the application clients. A business may have the different themes.

Integration tier: integrates different business process into a domain model. Domain data used in different business process can be integrated in this tier.

Resource tier: this is the tier contains the business data and also external resources such as mainframes systems, business-to-business (B2B) integration systems, and other services.

3.2.5. Establishment of services
A service is a function that is well defined, self-contained, and does not depend on the context or state of other services [SOA, 2004]. In software applications, the services are accessible to the components via a standard set of APIs.

Back to the architectural issues mentioned earlier on page 14, the architects have to handle resource lookups, data transaction and persistence, application security and deployment when designing the enterprise applications. To best deal with these issues regularly occurred in almost every enterprise application, architecture will provide a systematic base for the implementation of the design. The types of services used in enterprise application are component services, application services, platform services and hardware services.

3.3. J2EE overview

3.3.1. What is J2EE?

In the J2EE Specification [J2EE Specification, 2003], it defines a standard architecture composed of an application model, a platform for hosting applications, a Compatibility Test Suite (CTS) and a reference implementation.
The primary concern of J2EE is the platform specification: it describes the runtime environment for a J2EE application. This environment includes application components, containers, resource manager drivers, and databases. The elements of this environment communicate with a set of standard services that are also specified. The Figure 7 below shows the J2EE server model and in which tiers the J2EE components reside.

Figure 7. J2EE server model.

[Takagiwa et al., 2001]

3.3.2. What is J2EE technology?

From a pure technological point of view, three areas can be recognized (shown in Figure 8) within the J2EE specifications.
Component technologies: Software components are binary units of independent production, acquisition, and deployment that interact to form a functioning system [Szyperski, 1998]. The J2EE runtime environment defines the Application Client, Applet, Servlet and JavaServer Pages (Servlets/JSPs), and Enterprise Java Beans (EJB) as the four types of application components that a J2EE product must support. Each type of component is executed in a separated container. Figure 9 shows an overview of the application components in their respective containers along with the logic they usually handle in an enterprise application.
Services and service technologies: The services are functions that are accessible to the components via a standard set of APIs. For example, using the Java Naming Directory Interface (JNDI) APIs to access the naming services.

Communication technologies: The essence of J2EE is the definition of a distributed, object-oriented infrastructure. Components need to communicate with each other in this distributed world. Therefore, the containers need to provide the appropriate communication mechanisms to make this happen. An example of the communications included in the J2EE standards are RMI/IIOP protocols for remote method calls, Messaging technologies like JavaMail for programmatic access to e-mail and JMS for accessing messaging technologies, data format technologies like JAR files and XML.

3.4. HUVA system architecture

From the requirements in the previous chapter, the HUVA software should be widely accessible and easy to be adapted to different car service customers. In this part, the architecturally significant requirements of HUVA project are first defined. Then the layers, tiers and services are defined in HUVA, because HUVA is meant to be a J2EE product. At the end of this part, the HUVA architectural structure and relations are presented and explained.
3.4.1. HUVA architecturally significant requirements

The architecturally significant requirements (ASR) are the ones out of all requirements, in which the architect is most interested, and dominate the overall architecture [Kozaczynski, 2002]. For example the specific database architecture designed earlier in the legacy system can be considered to be an architectural significant requirement, because the new applications must cope with the previous database design. In HUVA, the ASRs can be classified into the following categories: performance, reliability, usability and adaptability.

- Performance

To cope with the accessibility and ease of use, HUVA is meant to be a web browser-enabled software, because every potential HUVA user will know how to use the web browser over the Internet or intranet. At this point, how to deal with the application throughout and response time to the request become significant. This also matches the requirement of the efficiency mentioned in the chapter 1 that the developing team has to make a software application faster and more efficient to use than the paper work delivered by the car service agents with proficiency. In practice, there are a couple of issues that have an impact on the HUVA performance. They are about HTTP session data management, application data transfer mechanisms, use of EJB components, resource pooling and caching.

A session in a browser presents that a conversation that spans multiple requests between a client and a host. The state of the session can be saved on either the client or the server. Handling the session data in HUVA in a proper way can increase the application speed. Saving session data on the client means that the session data is serialized and embedded within view HTML that is returned to the client. Saving session data on the server means that the session data is store on the server and retrieved by a session ID.

As the HUVA uses some key data that is currently stored within the enterprise’s legacy system, there will be data connections to the CD legacy system in addition to the new created data sources that store the reservation data. Which means that the data from the legacy system could be the row data. Data wrapping or parsing are likely to be needed. How to develop a data transfer mechanism handling the connection, data wrapping and transfer to the external data source, even the combination of different data sources, is the practical challenge the architect has to solve it in advance.

The EJB component in J2EE project is aimed to improve the system performance by handling the data persistence, system transaction, and security and so on in a standard industrial way. The use of EJB component has an impact on the data persistence and search performance.
Resource Pooling avoids making an expensive single call directly to the data source from the application. If a client is not currently using a server, that server’s precious resources can be returned to a pool to be reused when other clients connect. In HUVA, the resource pooling can include the sockets as well as the objects that live within the server. In HUVA, the database connection stuff should be considered to use a JDBC connection pool that avoids making a expensive single call directly to the data source from the JDBC Driver. In dealing with the transaction for business objects, pooling is also applied for EJB session and entity bean. In other words, the EJB session or entity bean can have an instance pool.

Caching is a proper idea to increase the performance is that the data can be kept around in the server’s memory and void costly network roundtrips and databases hits, as a lot of clients in real life share and make use of a common data category, like the data records generated by some common search.

- Reliability

The second architecturally significant requirement in HUVA was to make the system reliable. Two architectural issues (security and data persistence) are well considered in the effort to increase the system reliability.

In HUVA, the security matter is conducted to an acceptable level through the user of the following security mechanisms: authentication and authorization.

In distributed component computing, authentication is the mechanism by which callers and service providers prove to one another that they are acting on behalf of specific users or systems. For instance, the client may enter a username/password in web-based application, and those credentials are checked against the client information/profile stored in either a newly created database or legacy data storage. The idea about HUVA authentication is that HUVA takes the username and password as parameters to match the user info stored in the table in the legacy database, when a user logs in for the first time. Once the username and its password have been matched with the info in the legacy database, then the authentication is successful; and this authentication process must avoid the situation that the same user has to go through the authentication process over and over again when every time the user requests a resource.

The authorization mechanism limits interactions with resources to collections of users or systems for the purpose of enforcing integrity, confidentiality, or availability constraints. Two common types of authorization are declarative and programmatic authorization. In HUVA, the authorization should be done in association with business logic settings in user table in the
database. When a user has successfully logged into the application, the authentication is successfully completed. The user will see only the car manufacturer and service shops that the user is responsible for. The underlying idea should be that the program takes the predefined business settings (car manufacturer, service shops) in the user profile table as parameters to query the records that matched with the user’s responsibilities.

Data persistence is an important requirement, because the key information gained by HUVA booking process will be stored in the integrated CD system, even though HUVA modelled the real business process for making car service reservation and established its own objects. A key issue for initialising the HUVA project was that HUVA plays as a service extension of the CD ERP project. When the user browses the reservation information, the data related to the bookings is actually fetched from both the newly created database and the legacy CD system. Keeping the data used in HUVA and stored in CD persistent is the one of the key requirements. The actually data processing between HUVA and the legacy system CD is shown at Figure 10 as bellow:

---

**Figure 10.** Data connection between HUVA and the CD legacy system.

- Usability
Visualization is the use of computer-supported, interactive, visual representations of abstract data to amplify cognition [Card et al., 1999]. In the previous chapter, it was mentioned that the car service agent handles the booking on paper and also browses the information on paper. Especially, when the car service agents agree on the service date or time with the customers. The agents look at not only the existing reservations on the particular date but also the available workforces. At this point, HUVA must achieve the efficiency in browsing the reservation information. The visualization in HUVA means that the visual presentation of business data used in the car service reservation making process. How to visualize the booking information and also the workload of each group, team, or employee are the main issue in dealing with the software’s usability.

The solution for the information visualization in HUVA could be, for example, to design a well-considered data table or calendar that collects all the information from the reservations and also the workloads related the employees.

- Adaptability

In the previous chapter, it was mentioned that in addition to satisfying the committed customer, the HUVA was initially planned to target, in scope to domestic customers, auto-repairing shops in Finland. This requirement tells that HUVA software must be adaptable in the future so that it can fit different individual businesses. Functionally, even though HUVA is an extension of a part of service in an enterprise legacy system, HUVA is meant to be an independent and highly parameterised software system. At this point, HUVA should first provide the high flexibility for customizing its view; then HUVA should provide a certain mechanism for modifying its business model; finally HUVA must be independent of the enterprise legacy system, which is optional to use a set of stored procedure programs that act as an adapter to call the external data sources.

As the business at its customer side may develop with time being, the existing functions might be in need to be modified and new functions might be in need to be added. At this point, the HUVA software should be extensible for additional commands, models, or even the stored procedures programs to the legacy system.

3.4.2. HUVA layers, tiers, and services

According to the running environment mentioned in the previous chapter, the HUVA system is vertically divided into four layers, which are the application layer, the virtual platform layer, the upper platform layer and the lower
platform layer. As Figure 11 shows, the application components like EJB, Servlet/JSP components are running in the first layer. The APIs are stored in the second layer. The whole application and its service APIs are being deployed and running on the particular web application server, IBM WebSphere Web Application Server which is running on the IBM iSeries (OS/400) platform.

One important design step is to model the tiers that the application uses and to define the responsibilities of each tier. In general, there are only three tiers in HUVA, which are the presentation, business and integration tier. The Client tier is omitted, as the HUVA generates HTML pages to present the business information corresponding to the client’s request, and no additional client programs needed on client side, except for the client browser.

- The presentation tier

The main responsibility of the presentation tier is to present the car service reservation business information. As mentioned earlier, the business information must be rendered in a flexible and visualized way. The basic layout of the presentation tier is being constructed as a web calendar, which visualizes the real business schedules. With the additional associated information, the
final users can quickly observe not only the reservations and schedule but also
the supplementary information related to the schedule, reservation and even
the employee’s workload.

Other responsibilities at the presentation tier include the session data
management, client access control, view rendering and configuration, and
validation.

When working with the business tier, the presentation tier processes the
user’s request into a particular command and passes the command to the
business tier. In the meantime, the presentation tier redirects the user to the
result view where the result data responded from the business tier can be
captured. One thing to be mentioned here is that a controller layer resides
between presentation and application logic layer is used to manage the access
control and distribution of different requests throughout the whole application.

- The business tier

The business tier is responsible for almost all the application functionalities
related to the booking business, including: handling dynamic content
generation, implementing core application functionality, such as booking
processing and enforcing the application business rules, finally managing
transactions, such as by using JTA, and connection pooling for data access. In
particular, the business tier involves in the issue of using session and entity
bean. In HUVA, stateless session bean can be used. The entity beans or
customized data persistence mechanism can be used as coarse-gained business
components with the characteristics of persistence, transaction and long live.

- The integration tier

The main purpose of having this tier is to integrate and hold the persistent
application data in the external data sources. In HUVA, IBM DB2 Universal
database installed on the OS/400 platform is the database. JDBC Connectivity is
used as a database bridge to exchange data between business tier and
integration tier. In this tier, most work is done by the stored procedure
programs that reside on the OS/400 platform. The programs are called by RPG,
a fully procedural programming language, to provide data information needed
for making the response to the up-level request. This feature is defined due to
the previous database design in the legacy system.

The J2EE services used in HUVA are JNDI and JDBC. The purpose is to
locate the enterprise data source. The security and transaction are not handled
by JAAS and JTA, instead they are handled by the Servlet as a controller.
3.4.3. HUVA system structure

The following Figure 12 and 13 describes two types of the HUVA high-level system structure. The main difference between these two types is that the use of EJB Component.

For the type shown at Figure 12, the whole application architecture follows the J2EE tiered approach and is divided into the three tiers. They are presentation, business and integration tier. The client tier and resource tier are not considered at the HUVA architecture.

![HUVA high-level system structure](image)

Figure 12. HUVA high-level system structure.

For the HUVA system structure shown at Figure 12, the Servlet and JSPs are used to the web client request. When the client makes a request to HUVA, the Servlet handles the request and deliver the request to the business access interface component. In the meantime, the Servlet dispatches the corresponding JSP view to the user. The JSP view gets the view content returned by the business access interface component which calls the business services provides by the business tier.

In the business tier, a session EJB component is used to present the business process objects. The session beans in HUVA implements the real car reservation logics, rules, algorithms and workflow. For example, the session beans can
perform the service reservation making, updating and searching. The session EJB is a relatively short-lived component. It has roughly the lifetime equivalent of a session or a lifetime of a client call.

The component down after the Session EJB is the persistence component. The purpose of having this persistence component is to keep the business data persistent from all the activities. There can be a lot of specific strategies when this issue comes to the implementation level. For instances, one option is that no object model is used (read and write data directly to the data resource for each CRUD business operation [FOWLER, 2002]), the other option is to use entity beans (CMP or PMP). After the persistence component, HUVA exchanges the data with two data sources. One is the newly created HUVA database; the other is the data storage in the CD legacy system.

In the integration tier, the stored procedure component is used to communicate with the external data source. In HUVA, the external data source is stored in the legacy system; there are a lot of other applications resided within the legacy system. The stored procedure component was previously required to be used in cope with the legacy system.

In addition to use of EJB components, the HUVA has been designed without the use of the EJB. Figure 13 below shows an alternative HUVA architecture design.

---

Figure 13. HUVA alternative system structure.
This alternative architectural design does not consider the tiered approach; it uses only one web server container to hold the whole application. When the client request comes via HTTP, the Servlet acts as a controller that calls the business model implemented by a Java class and dispatches a corresponding JSP view to the client. The view gets the business content from the model that returns the data after being called the controller.

When talking the data sources, HUVA uses two different components. A data access component is used to access the newly created HUVA database, while a stored procedure component is used to exchange the data between HUVA and the CD legacy system.

Comparing the architectural design shown at Figure 13 to Figure 12, the later one discards the tiered approach and implements all HUVA business model objects as Java classes, while the first one breaks the system into three tiers and the business model is handled by the use of EJB components. To access the data sources, the alternative HUVA architecture uses two components (data access component and stored procedure component) to handle the data call and persistence. With the first design, the system increases flexibility, adaptability and performance, but also increases the work in its implementation and requires the use of an EJB container. The second design reduces the work amount during the development, but it increases the complexity of the business model implemented by Java classes and decreases the system flexibility and adaptability.

In practice, the first architecture design shown at Figure 12 was chosen to be implemented. But the second design was kept as an alternative. In the next chapter, the design patterns will be discussed to elaborate the first HUVA architectural design.
4. Application design patterns

In this chapter, the main purpose is to apply the J2EE patterns to the previously defined HUVA architecture so that the design is further deepened. In the arrangement, after short pattern introduction given first, the J2EE patterns overview is presented according to the tiers. For the rest of the chapter, the J2EE patterns used in HUVA are presented and some design considerations in the each tier are also discussed. The J2EE patterns discussed here are originally from Sun’s blueprints [Sun BluePrints, 2004] and some of them are built on the basis of some well-known patterns described in literatures such as Design Patterns: Elements of Reusable Object-Oriented Software [Gamma et al., 1994], and Patterns-Oriented of Software Architecture [POSA 1996, POSA 2000].

4.1. Introduction to pattern

In software engineering, people have been making different efforts to design robust and reusable software. However, the process of designing such a software system is not easy. More and more people share not only their particular successful experiences with each other but also the difficulties they are facing. Pattern is the term they use to communicate with each other whom might not participate in the situation, but understand the problem in a context.

Historically, pattern in civil engineering was first documented in Christopher Alexander’s books [Alexander 1977, Alexander 1979]. In A Pattern Language (1977), Alexander writes:

"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of solution to that problem, in such a way that you can use this solution a million time over, without ever doing it the same way twice."

In the book, Christopher Alexander also gave the pattern a definition, which has been well known by the people in software engineering:

“Each Pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution.”

Later, pattern in software engineering was popularised by the book: Design Patterns: Elements of Reusable Object-Oriented Software by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (also known as the Gang of Four). The idea of patterns [Gamma et al., 1994] in software design and construction is to capture design experience in a form that people can use effectively. With patterns, the system design is improved. Because a pattern means a proven solution in a context; can be applied to any level of the design;
and provides a common vocabulary to communicate among the different roles in a project.

4.2. J2EE patterns overview

J2EE patterns are a collection of J2EE-based solutions to common problems, and reflect the collective expertise and experience of Java technology architects over the past years [Alur et al., 2003]. These J2EE patterns are typically addressing to the problems encountered in building enterprise applications, especially with regards to the J2EE components such as Servlets, JavaServer Pages (JSP), and Enterprise Java Beans (EJB). The J2EE patterns are collected with an approach of best practices, and are presenting in a way that enables people to apply the patterns to theorize own particular applications. Each pattern hovers somewhere between a design pattern and an architectural pattern [Alur et al., 2003]. Through the tiered approach, J2EE patterns are clarified according to functionality into three logical architectural tiers: presentation, business and integration tier. The patterns are used in combination to solve common problems.

The J2EE patterns discussed here are originally from Sun’s blueprints [Sun BluePrints, 2004]. However it is easy to notice that sometimes some patterns are built on the basis of some well-known patterns described in literatures such as Design Patterns: Elements of Reusable Object-Oriented Software [Gamma et al., 1994], and Patterns-Oriented of Software Architecture [POSA 1996, POSA, 2000].

4.2.1. Presentation tier patterns

In the chapter 3, the presentation tier was mentioned that the presentation tier encapsulates all presentation logic required to serve the clients that access the system. The responsibilities of the presentation tier are providing sign-on, conducting session management, controlling access to the business services, and managing the responses to the client. In J2EE, the presentation tier is mainly implemented with JSPs/Servlets.

The J2EE patterns that can be applied to this tier are Intercepting Filter, Front Controller, Context Object, Application Controller, View Helper, Composite View, Service to Worker, and Dispatch View [Alur et al., 2003].

More specifically, the patterns such as Intercepting Filter, Front Controller and Application Controller are used to centralize control for the client request handling. The Context Object is used to reduce the coupling between the protocol-specific system information and the application components. The View Helper and Composite View are used to handle the multiple views with different data. A combination of different patterns, the Service to Worker and
Dispatch View are similar patterns used to handle the requests and dispatch the corresponding views. The main difference between the two patterns is that Service to Worker handles the activities before a view is dispatched, while the Dispatch View leaves many activities to the view processing.


4.2.2. Business tier patterns

The patterns applied to the business tier are meant to handle the business process including both the business logic and business data. It is common that most business processing for the application is centralized in this tier and Enterprise Java Bean components are the usual and preferred solution for implementing the business objects in the business tier.

The J2EE patterns, which can be applied to this tier, are Business Delegate, Service Locator, Session Façade, Application Service, Business Object, Composite Entity, Transfer Object, Transfer Object Assembler, and Value List Handler [Alur et al., 2003].

The Business Delegate and Session Façade are used to encapsulate the access to services. Service Locator is used to encapsulate the service lookups. Transfer Object, Transfer Object Assembler and Value List Handler are used to handle the data. Business Object and Composite Entity are used to deal with business model.


4.2.3. Integration tier patterns

The business tier is coupled with the integration tier whenever the business tier has the need to access the data sources or services that reside in the integration tier. Consequently this tier is dealing with the communication between the
business tier and the external data sources. The Technologies like JDBC or J2EE connector technology are used in this tier.

The J2EE patterns, which can be applied to this tier, are Data Access Object, Service Activator, Domain Store, and Web Service Broker [Alur et al., 2003].


4.2.4. J2EE patterns relationships

Individual patterns provide the solution for a particular problem in a context. If every pattern can only exist in isolation, then the patterns will become useless. How to apply different patterns together into an application to get a larger solution is becoming a major issue in the design.

The J2EE patterns are meant to work together to solve multiple problems and build larger and complete enterprise systems. Figure 14 shows the relationships among the patterns.
Figure 14. Sun J2EE pattern relationships.

[Alur et al., 2003]

As the figure 14 shows, Intercepting Filter acts as filter to the incoming requests and outgoing responses. Front Controller processes the presentation logic for an instance the JSPs and Servlets are handled in the controller. The View Helper can access the business services and separates the form-related view from the business logic so that numerous views can be constructed from
the Composite View pattern without caring of the business logic. The Service to Worker and Dispatcher View are two complete patterns for the presentation tier as they combined the controller, view and helper to work together to build the presentation tier.

Business Delegate reduces coupling between different tiers and provides an entry point for accessing the services in the business tier. Service Locator centralizes the lookup work for resources and can be used by Business Delegate and other clients. Session Façade is just like an interface providing coarse-grained services to the clients by hiding its implementation complexities. The business domain models are implemented with the Business Object that separates the business data and logic. The Composite Entity implements the Business Object using local Entity Beans and Java objects. The Data Access Objects can be used to facilitate the data persistence while Composite Entity is used. A powerful pattern to reduce the network overhead is the Transfer Object that can be constructed and reused to transfer the data across different tiers.

In the integration tier, Data Access Object connects the resource tier from the business tier. It abstracts and encapsulates all the access to the persistent store. The Data Access Object can be used either by the pattern Domain Store, which implements transparent persistence for the business object model, or by the Composite Entity to facilitate the persistence when implemented with bean-managed persistence.

4.3. HUVA presentation tier patterns
In HUVA, the responsibilities of the presentation tier are providing sign-on, managing the session data, controlling the data access to business service, and generating the responding views to the clients.

4.3.1. Presentation tier design considerations
In this part, some high level design issues addressed in the previous chapter will be further detailed so that they can be implemented in practice. However, the practical implementation details will not be covered here.

- Session Management

Typically, there are three strategies for saving session state on client. One is to use HTML hidden fields, the second is to use http cookies and the third is to directly embed the state into the Uniform Resource Identifiers (URIs) referenced in the HTML page. However saving the session data on client has some drawbacks, e.g. only limited data can be stored and string is the only data type that can be stored. Compared to saving the session data on client, saving the data on the server is preferable for application with large amount of session data. The advantage is that there are no size and type limitation.
In HUVA application, session data is handle by session beans. The state is stored in a Java bean object on the Web Server. The state info can be retrieved throughout a user session unless the session is timeout. It was determined by the scope of HUVA project, because all the clients will access HUVA through the client’s HTTP Server, more specifically either through a Servlet or JSP. Figure 15 shows that the session state is stored in presentation tier.

![Figure 15. Storing the session state in presentation tier.](image)

- **Controlling Client Access**

In HUVA, the security matter is conducted to an acceptable level through the authentication and authorization.

In HUVA, the authentication is done during the sign on process. A form-based authentication is used at the front point where a user accesses the HUVA home page. Form-based authentication lets developers customize the authentication user interface presented by an HTTP browser. The content of the user dialog (such as, username and password) is modified with some security rules and sent as plain text. The idea about HUVA authentication is that HUVA takes the username and password as parameters to match the user info stored in the table in DB2 database, when a user is first time to login HUVA. Once the username and its password have been matched with the info in the database, then the authentication is successful and meanwhile the HUVA generates HTTP cookies to user’s client machine. The use of HTTP cookies avoids the situation that the same user has to go through the authentication process over and over again, every time the user requests a resource. Because the authentication checks the hash values in HTTP cookies on the client machine if the user is a valid session when the user is second time to access. Figure 16 shows the overview of the HUVA authentication process. In addition, an XML file is used to configure the whole process, and a Java class for example named LoginPerson.class is used for holding the session data (Person ID, Modified time and so on).
The authorization mechanism limits interactions with resources to collections of users or systems for the purpose of enforcing integrity, confidentiality, and availability constraints. Two common types of authorization are declarative
and programmatic authorization. In HUVA, the authorization is done in association with business logic settings in a user table in the database. For an instance when a user is successfully logged into the application, then the user will see only the car manufacturer and service shops that the user is responsible for. The underlying idea is that the program takes the predefined business settings (car manufacturer, service shops) in a user profile table as parameters to query the records that matched with the user’s responsibilities. In this way, HUVA authorization should belong to the type of programmatic authorization.

- Avoiding duplicated submissions

In a web browser environment, the user can click the back button to return to the previous page and re-submit the form data again. This re-submission causes a duplicate transaction and incorrect data. To avoid this duplicated submission, in HUVA a synchronizer token is used in the front controller. A synchronizer token is set in the user session and included into each form returned by the client. When the form is first submitted, the tokens will match and the token in the user’s session will be modified and also included to the next form that will be returned by the client as well. When the form is re-submitted, the two tokens will not match because the token in the user’s session was modified when the form was first submitted.

![Figure 17. HUVA synchronizer token.](image)

Figure 17 shows the synchronizer token in the interaction between the client and the presentation tier.

- Performing validation
Validation performance can be done either on the client or the server, and even on both. In HUVA, the validation check is done both on the client and server. More specifically, the validation check is implemented on the base of abstract types, which means there is a common validation checking mechanism that is setup on server and included in each form returned by the client.

### 4.3.2. Service to worker

In HUVA, if the user such as a car service agent wants to browse the existing reservations; the user will have options to view different sub pages containing different information related to the reservation. The web page flow is not static, instead rather dynamic. Of course, the business logic is executed according to the user’s specific request. So before a responding view is rendered, the specific business logic is executed to serve a request in order to retrieve content that will be used to generate dynamic response.

The solution is to use service to worker in HUVA to centralize control and request handling to retrieve a presentation model before turning control over to the view. The view generates a dynamic response based on the presentation model.

---

**Figure 18. HUVA service to worker.**

In HUVA, the Service to Worker is combination of several other patterns. As shown at Figure 18, the HUVAFrontController acts as a Servlet and provides a centralized entry point for handling requests. The HUVAApplicationController maps the incoming request parameters to specific-processing classes, and to view components that handle each request. In particular, the HUVACommand pattern handles the specific parameters
received from the HUVAApplicationController and accesses the HUVABusinessServices accordingly. The HUVAPresentationModel is returned when the HUVABusinessServices is invoked via the command. The ViewHeper pattern helps the view adapt and transform the HUVAPresentationModel to the user. Figure 19 as below shows the sequence diagram.
Figure 19. Sequence diagram of HUVA service to worker.
With the use of this Server to Worker, the modularity and reusability of the whole HUVA system are improved by centralizing control and request-handling logic. The command and view processing code can be reused. New commands and corresponding views can be added easily in a similar way.

Figure 20 below shows the use of Front Controller and Application Controller in HUVA. The idea is that the HUVAFrontController as a Servlet to handle the request from the client and HUVAApplicationController is used to handle the action and view.

![Figure 20. HUVA front controller and application controller.](image)

In this way, all the requests from the client are handled at the initial point, HUVAFrontController; and both the command and view management are done by the HUVAApplicationController through some mechanism such as mapping.

The commands in the HUVA Service to Worker pattern are implemented under the following command framework (shown at Figure 21). In general there are two types of commands: one is the command responsible for redirecting the clients and named as HUVARedirectCommand, the other is the command responsible for preparing the corresponding view and named as HUVAViewCommand. Both types are inherited from an abstract command named HUVACommand which implements the Command interface. A value object named CommandReturnAttributes is used to store the values returned by all the commands. For the HUVARedirectCommand, the redirect info can be customized in association with the CommandParameter.
In HUVA, the views are separated from the processing logic. It is due to the use of View Helper pattern. The idea of View Helper is that a view delegates its processing responsibilities to its helper classes, which can be implemented as Java bean, or custom tags. The View Helper acts as an adapter between the view and the model, performing the transformation logic, such as generating different customized tables for different data. In HUVA, the helper classes are implemented as Java beans. The figure 22 below shows the structure:
According to the figure 22, the client dispatches a view that uses the helper bean to adapt the model data provided by command processing.

4.4. HUVA business Tier patterns

In HUVA, the responsibilities of the business tier are providing business services such as booking processing and information search.

4.4.1. Business tier design considerations

- Using Session Beans

According to the EJB 2.0 specification [EJB Specification, 2001], session beans are business process objects. They implement business logic, business rules, algorithms, and workflow. Compared to entity beans, a session bean is a relatively short-lived component from the point of lifetime. Session beans have two subtypes: stateful and stateless. A stateless session bean does not hold any conversational state. At this point, a stateless can be free for reuse once the client’s invocation on the session bean is finished. The EJB container can maintain a pool of session beans and to use them over and over again on the client’s invocation. In HUVA, stateful session are not needed, instead the stateless session are used to implement the business logics, for example, getting daily car reservations.

- Using Entity Beans

The entity beans are permanent persistent objects modelling the business fundamentals. The appropriate use of an entity bean is as a distributed, shared, transactional, and persistent object [Alur et al., 2003]. For the entity bean, the EJB containers and provide other infrastructural support to the system.
attributes such as scalability, security and performance. In HUVA, the entity bean is not used, because HUVA is not a large project in scale and some operations in HUVA are limited to the legacy data source. Without the use of the entity bean, the HUVA can gain the data control manually. The returned data types by the session bean are suited for tabular business data, while the returned data types by the entity bean are Java objects.

In the HUVA business tier, the J2ee patterns like Service Locator, Business Delegate, Session Façade, Business Object, and Transfer Object are used to implement the booking process. Figure 23 shows the pattern structure.

4.4.2. Business delegate and session façade

In HUVA, the booking service components are hidden from the client. It means that in practice the booking service code can be changed without the change to the client code. Another significant reason to prevent the client from directly accessing the booking service is that the service lookup such as JNDI can be centralized, because otherwise the service lookup will have to be embedded in different client calls.
The Business Delegate and Session Façade are used to separate the client from accessing the booking component directly, and to centralize the service lookup and the booking business logic. Figure 24 shows that when the client tries to access the HUVA booking business services, the HUVABusinessDelegate uses the common HUVAServiceLocator to lookup the service and then delegate the request to it. However, the request is not directly delegated to the specific business object, instead it is delegated to the HUVASessionFacade which acts as the booking service interface. The HUVASessionFacade is used to decouple the relation between the client and particular booking business objects. It hides the real implementation of the particular business logic and exposes only the required interface to the client.

Figure 24. HUVA business delegate and session façade.
4.4.3. Service locator

In HUVA, a Service Locator pattern is used to implement the data source lookup. Figure 25 shows, the client first gets an instance of service locator, which creates the initialContext and Cache. Then the service locator looks up the JDBC data source from the initialContext through the JNDIRegistry. Once the data source is found, the data source is put to the Cache from the initialContext. Next time, when the client tries to get the same data source again, the service locator will get the data source from the Cache.
4.4.4. Business object and transfer object

According to the Sun blueprints [Sun BluePrints, 2004], business objects encapsulate and manage the business data, behaviour and persistence, while the transfer objects carry multiple data elements across a tier.

Business objects implement a reusable layer of business entities that describe a business domain. The business objects use the transfer objects to carry their values. A transfer object can contain all the data elements, instead of
individual data elements. According to the specific situation, a business object can creates multiple transfer objects. Figure 26 shows the business object and transfer object.

![Figure 26. HUVA Business object and transfer object.](image)

### 4.5. HUVA integration tier patterns

In HUVA, the responsibility of the integration tier is to manage the communication with the external data source, in particular the external CD legacy data storage.

#### 4.5.1. Integration tier design considerations

- Data persistence

  Data persistence in enterprise system means that how to store the data in a permanent data storage and how to access it later on over and over again. A persistence layer means that a collection of classes that provide business objects an ability to be persistent. The persistence layer effectively wraps the
persistence mechanism. In general, the persistence layer (shown at figure 27) exists somewhere between the domain objects and database.

In HUVA, the persistence layer is physically placed in the business tier, but it is closely connected with the integration tier. This persistence layer manages the persistent data not only from the HUVA database but also from the CD legacy system. Figure 28 shows the HUVA persistence layer and its relation with the data sources.
4.5.2. **Data access object**

In HUVA, the business data is stored both in the HUVA database and the CD legacy system, there must be a data access mechanism to access and manipulate data in such persistent storages. And also there must be a set of uniformed data access APIs provided by the mechanism for various types of data sources, such as the IBM DB2, a Relational Database Management System (RDBMS).

The solution is to use Data Access Object to abstract and encapsulate all access to the persistent store. The data Access Object manages the connection to the data source for retrieving and storing the data.
The Data Access Object discards the type of the data source used; it always provides a uniform API to its client. In HUVA, the Data Access Object encapsulates all the JDBC usage inside the access mechanism and does not expose any exception, data structure, object, or interface that belongs to the data query package such as java.sql.* or javax.sql.* to the client. Furthermore, whenever the data source implementation changes, the interface exposed by the Data Access Objects to the client will not change. Figure 29 shows this J2EE pattern.
As the above sequence diagram (Figure 30) shows, the client first creates the HuvaDAO that looks up the HuvaDataSource. When the client sends the request to get the data, the HuvaDAO opens a connection to HuvaDataSource and executes the query. After the query is executed, a ResultSet is generated and returned back to the HuvaDAO. What the HuvaDAO does is that it creates a transfer object, then sends the returned data to the transfer object and closes the connection to the HuvaDataSource. Finally what the client gets by calling the HuvaDAO is the returned transfer object containing all the data from the HuvaDataSource.
4.5.3. Domain store

Concerning the sophisticated persistence strategies required by the business object model, the system architects or developers have options to choose whether the data persistence is managed by the EJB component. For example, Container Managed Persistence (CMP) or Bean Managed Persistence (BMP), or by a separate persistence mechanism that separates the persistence from the business object model.

In HUVA, the persistence is not put into the object models implemented as the business objects. The entity beans are also not used. Especially, due to the system adaptability, the one of architecture requirements, HUVA application should achieve the flexibility of either running in both web and EJB container or running in a web container only. Considering all the above design issues, a Domain Store pattern is used to implement the persistence that separates from the business objects.

As Figure 31 shows, the HUVAController does not interact with the HUVABusinessObject directly; instead it interacts with the HUVABookingHandler to update the business objects. The HUVABookingHandler actually handles the persistent state of the business objects and enforces transactional retrieval and storage of state from the data source via the HUVADataAccessObject. If there is nothing involved in the persistent state of business object, the HUVAController can call to the HUVADataAccessObject directly for accessing the data source.
5. System review

So far, the HUVA system requirements and the available technologies have been presented and analyzed; a type of J2EE architecture has been presented; and the possible J2EE design patterns used to elaborate the architectural design have been also discussed. This part of the thesis is to review the whole project.

HUVA was a project implementing a business model for making car service reservations. In the context, this car service reservation model was an extension of the ERP system, Solteq CD system, even though HUVA was setup as a brand new project. To achieve the competitiveness in practice, HUVA was required to be used efficiently by the end users, well graphically visualized for the information presentation, available for the multiple car shops distributed in different locations, flexible for the customization and modification, and reusable in the data collection.

The technologies available for the development in practice were mainly the Java technologies based on Java Platform 2, Enterprise Edition (J2EE), such as JavaServer Pages (JSP), Servlet, Enterprise JavaBeans (EJBs), Java Database Connectivity (JDBC) and Java Naming and Directory Interface (JNDI). The running environment was supported by the IBM’s WebSphere Application Server and OS/400 Operation system.

Compared to the traditional waterfall [Cantor, 2002] process in software development, the HUVa development process is managed by the Rational Unified Process (RUP). A typical iteration [Pollice, 2003] flow has been used.
As Figure 32 shows, the project was initially planned by the Solteq Car Division; the car reservation process was modeled and the system requirements were defined by the project management team; the Java development team designed the system and implemented the system according to the design. So far, a testing version of HUVA system has been deployed at the customer’s premise for the testing purpose. Further evaluation will be done later after the testing phase. In turn, the business model could be improved correspondingly if the later evaluation shows the need.

By modeling the car service reservation business, the HUVA system architectural requirements (shown at table 1) were defined as performance, reliability, usability and adaptability.

<table>
<thead>
<tr>
<th>Description</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance</td>
<td>Response time and network overhead</td>
</tr>
<tr>
<td>Reliability</td>
<td>Security and data persistence</td>
</tr>
<tr>
<td>Usability</td>
<td>Information visualization and parameterizations</td>
</tr>
<tr>
<td>Adaptability</td>
<td>Extensibility and migration</td>
</tr>
</tbody>
</table>

In the architectural design, alone with the J2EE tiered architecture, the HUVA application was divided into three tiers, which are the HUVA
presentation tier, the HUVA business tier and the HUVA integration tier. The HUVA presentation tier is mainly responsible for the presentation of the car service reservation. The HUVA business tier is mainly responsible for the online booking processing and information related search. And the HUVA integration tier is mainly responsible for the communication with the external data source.

Figure 33. HUVA architecture.

To elaborate the above architecture, the J2EE patterns are used to provide the proven solutions to the recurring problems in such a kind of J2EE architecture. The main patterns used throughout the HUVA application are Service to Worker, Business Delegate, Session Façade, Business Object, Transfer Object, Domain Store and Data Access Object. Figure 34 shows the patterns and their relationships.
Above all, the HUVA architecture defines the HUVA system structure and the relation; the use of J2EE patterns provided the proven solutions to the
recurring problems in the system. In practice, the HUVA was designed in much more details for the implementation by applying the top-down approach. After deploying the application into the real business situation, the system can be tested and evaluated. According the iteration flow, the existing business model can be updated and the software can be adapted to fit the business demand that changes dramatically in real life.
6. Summary

This thesis describes an enterprise software system in which the J2EE architecture and patterns are applied. The thesis starts from the application domain, the car service reservation in Finland, goes through the analysis process and ends at the application design phase.

Within the thesis, the car service reservation business was modelled and the system requirements were defined with the respects to the customer needs, the extension of enterprise legacy system, and the company development environment and technologies available.

As theoretical parts, software architecture and patterns are introduced in brief; the J2EE, as an industrial standardization, was introduced from its architecture and technology perspectives. Based on the theoretical parts, the HUVA application was designed at a high level. The system architecture and its alternative were first defined. The current architecture obeyed the J2EE tiered architecture. The application was divided into three tiers (presentation tier, business tier and integration tier) according to the distributed responsibilities; and Web and EJB container were required. In the alternative architecture, the tiered approach was not considered and only the Web container was required in the running environment, instead of the two types of the container.

After the definition of the HUVA architecture, J2EE patterns that stand between the architectural patterns and design patterns were used to elaborate the architectural design and to provide the proven solutions to the recurring problems in the context of HUVA realization. One thing to be mentioned is that the J2EE software developers, in real life, have their own options to use the different J2EE patterns to solve their special problems in the particular context. Nowadays more and more J2EE patterns are even being created and documented by software developers.

With the development of enterprise software, the current J2EE architecture and patterns will be much widely applied or adapted to build robust enterprise systems. The new Java related technologies will be invented and included into the J2EE; and the new proven ways of using such technologies will be shared with all more and more participants in the area. All of these together will evolve the J2EE.
References


[Gamma et al., 1994] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley, 1994.


http://www.service-architecture.com/web-services/articles/service-oriented_architecture_soa_definition.html

Available at http://java.sun.com/reference/blueprints

