# CHAPTER 5: IMPLEMENTATION

## 5.1 System Implementation

This is the process of converting the system design into an executable system.

Travel Management is designed in such a way as a fully functioning web based solution, which developed under the J2EE technology. It is capable of running on different platforms such as Linux, Windows etc, since it has developed under the facilities of platform independent architecture in J2EE technology.

- For the development of user interfaces, which runs as web pages in the web browser are developed as .jsp pages and IntelJ and Macromedia Dreamviewer are the tools used for this purpose.
- Well Know open source application server, JBoss 4.0.3 is the application server, which responsible for the middle layer server processor activities.
- Oracle 10g is the database server used in the development stage of the application. Since the systems persistence layer is consist of Data Access Object, which is an important component in J2EE technology, is capable of switching form one database platform to another depending on the user preference.
- Intelj is the IDE tool user for the implementation of classes and Enterprise Java Beans.



## 5.2 System Architecture

Main architecture of the system is based upon the well-known design pattern named Model View Controller Architecture (MVC).

### 5.2.1 Model-View-Controller Architecture

The Model-View-Controller architecture is a widely used architectural approach for interactive applications. It divides functionality among objects involved in maintaining

and presenting data to minimize the degree of coupling between the objects. The architecture maps traditional application tasks--input, processing, and output--to the graphical user interaction model. They also map into the domain of multitier Web-based enterprise applications.

The MVC architecture divides applications into three layers--model, view, and controller--and decouples their respective responsibilities. Each layer handles specific tasks and has specific responsibilities to the other areas.[4]

- A ***model*** represents business data and business logic or operations that govern access and modification of this business data. Often the model serves as a software approximation to real-world functionality. The model notifies views when it changes and provides the ability for the view to query the model about its state. It also provides the ability for the controller to access application functionality encapsulated by the model.
- A ***view*** renders the contents of a model. It accesses data from the model and specifies how that data should be presented. It updates data presentation when the model changes. A view also forwards user input to a controller.
- A ***controller*** defines application behavior. It dispatches user requests and selects views for presentation. It interprets user inputs and maps them into actions to be performed by the model. In a stand-alone GUI client, user inputs include button clicks and menu selections. In a Web application, they are HTTP GET and POST requests to the Web tier. A controller selects the next view to display based on the user interactions and the outcome of the model operations. An application typically has one controller for each set of related functionality. Some applications use a separate controller for each client type, because view interaction and selection often vary between client types.

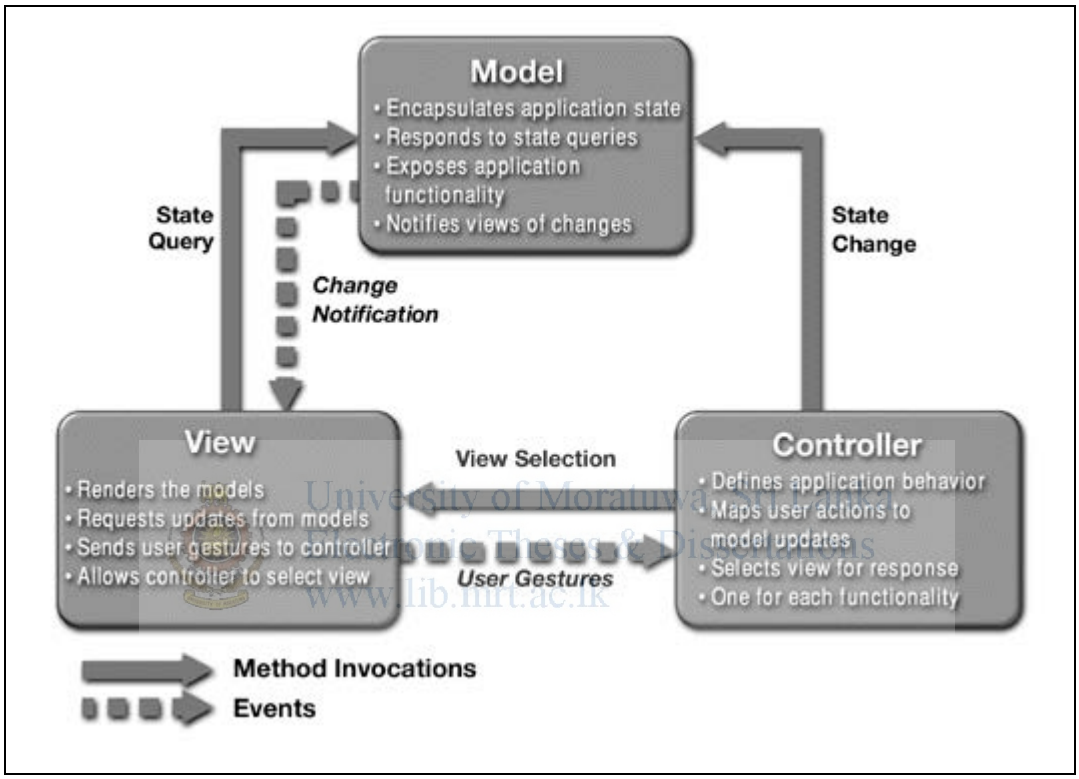


Separating responsibilities among model, view, and controller objects reduces code duplication and makes applications easier to maintain. It also makes handling data easier, whether adding new data sources or changing data presentation, because business logic is

kept separate from data. It is easier to support new client types, because it is not necessary to change the business logic with the addition of each new type of client.

Following figure depicts the relationships between the model, view, and controller layers of an MVC application

**Figure 5-1 Model-View-Controller Architecture**



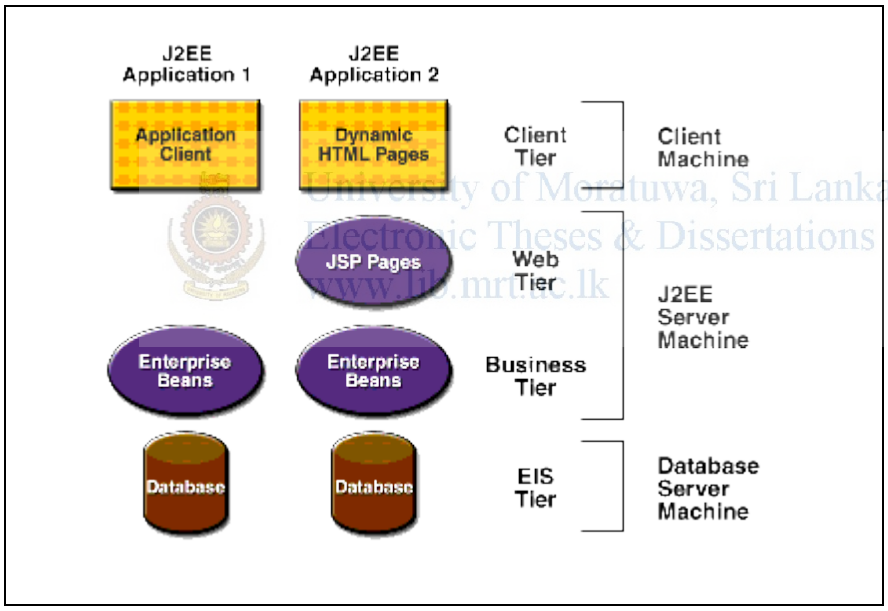

By following the above-mentioned architecture, the final outcome of the system is multilayered J2EE application, which runs on a J2EE application server. Following paragraph will describe the components in a three-tried architecture J2EE application by illustrating a sample model view. Beyond that it will describe the actual implementation of the **Travel Manager System** with the actual classes and components in it compared to the sample model.

### 5.2.2 Sample Multitier Application Model

The J2EE platform uses a distributed multitiered application model for enterprise applications. Figure 5.2 shows two multitiered J2EE application models divided into the three and four tiers as described in the following list.

- Client-tier components run on the client machine
- Web-tier components run on the J2EE server.
- Business-tier components run on the J2EE server
- Persistent-tier components run on the database server.

**Figure 5-2 Multitier Application Models**





According to the illustration showing in the above figures, the multitier architecture is applied for the system. The client tier is consisting of JSP pages which servers as dynamic web pages. These dynamic web pages are ideal for a system like this since the need of displaying user views using dynamic data sets. The middle tier or the business tier is consisting of EJB classes with the assistance of the controller servlet class. The persistence layer is the database layer users to store the travel data need for the system.

### 5.2.3 Design Patterns

#### 5.2.3.1 What is a design pattern

Can be thought of as a best practice solution to a common, recurring problem. [4], [5]Consists of:

- Name
- Description of problem and its context
- Solution
- Consequences of use

#### 5.2.3.2 J2EE Design Patterns

For J2EE, we instead group the patterns based on the tier they are associated with:

- Data Tier
- Business Tier
- Web Tier



#### 5.2.3.3 Data Tier Patterns

Data Access Object (DAO)

#### 5.2.3.4 What Is a Data Access Object?

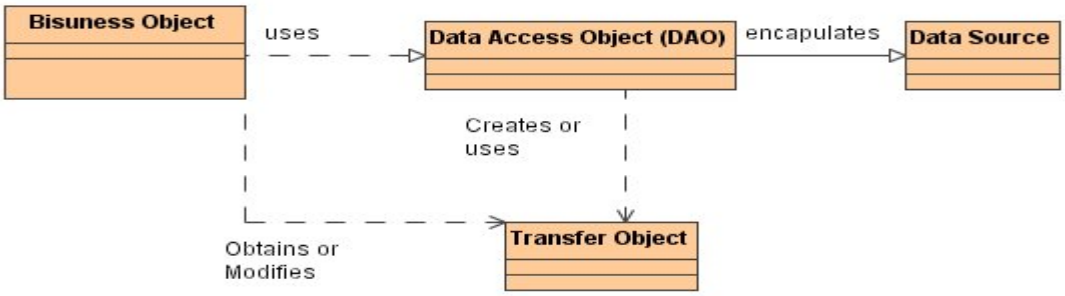
Object that encapsulates access to persistent storage
E.g. RDBMS, LDAP, etc.
Decouples client interface from underlying data access mechanics
e.g. Manages connection, adapts APIs

#### 5.2.3.5 How Does a DAO Work?

Abstracts the process of managing and accessing the connection to the data source
Encapsulates any proprietary APIs
e.g. Translates lower level exceptions

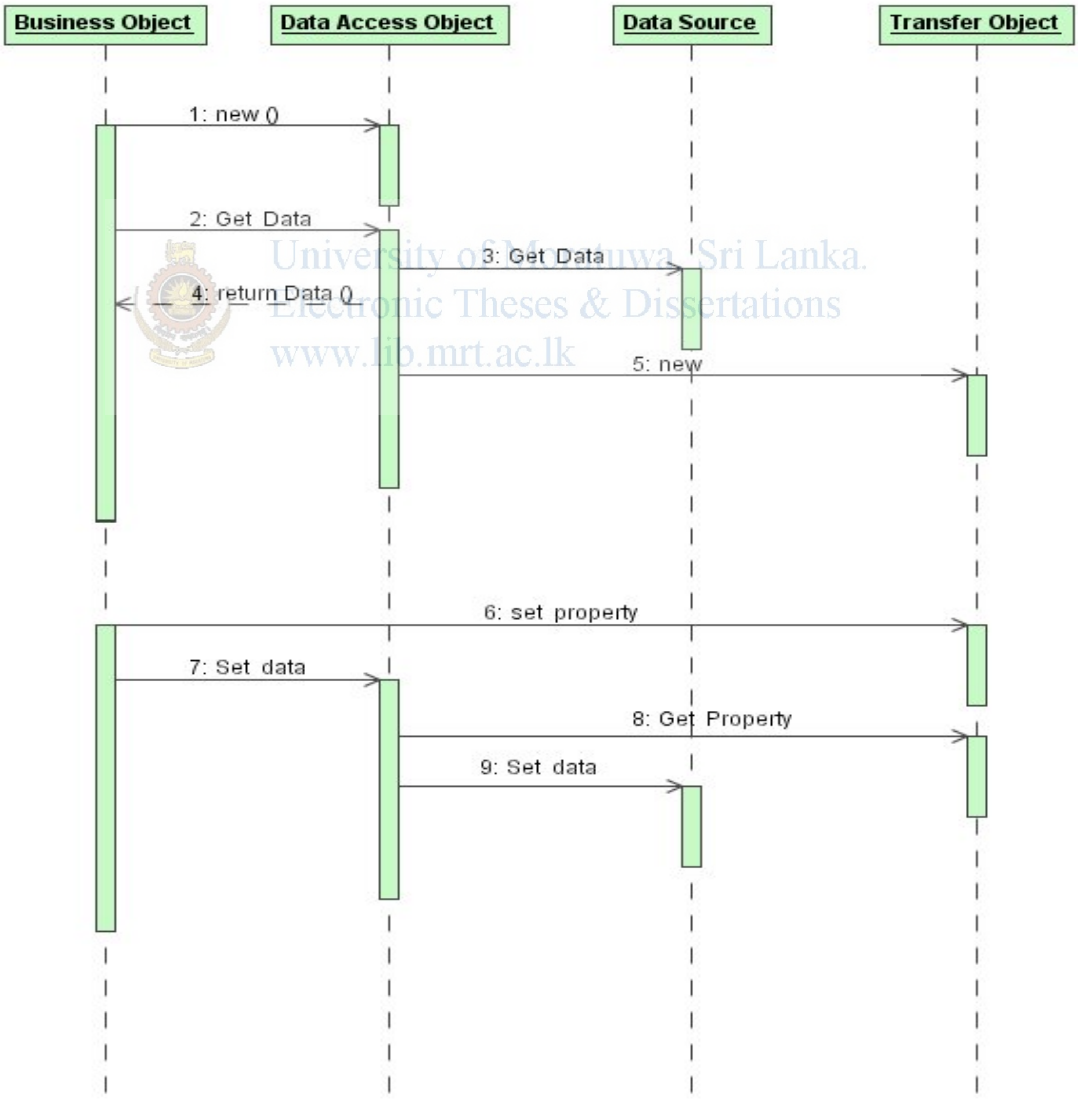
### 5.2.4 DAO Relationships

**Figure 5-3 DAO Relationships**




#### 5.2.4.1 DAO Interactions

**Figure 5-4 DAO Interaction**



#### 5.2.4.2 DAO Advantages

- Abstracts implementation details
- Data source independent
- Enables easier migration
- Encapsulates proprietary APIs
- Centralizes all data access into a single layer
- Reduces code complexity

#### 5.2.4.3 Disadvantages of the DAO

- Not useful for CMP EJBs
- Can still be used with the Fast Lane Reader pattern
- Adds an extra layer to architecture

#### 5.2.4.4 Why Use a DAO?



- Allows the underlying data access mechanism to change independently of the client code
- Allows migration of the data source
- Reduces the code complexity of the business objects
- Application is easier to manage and maintain
- All data access is centralized into a separate layer

### 5.2.5 Business Tier Patterns

- Transfer Object
- Service Locator
- Session Façade
- Business Delegate
- Fast Lane Reader

#### 5.2.5.1 What Is a Transfer Object?

- A Transfer Object is a data envelope used to transfer groups of related attributes between application tiers
- Useful when the client must use RMI to retrieve and update data
- Not as useful for communication between beans that can use local interfaces (EJB 2.0)
- Can be used to transfer arbitrary sets of data (Custom Transfer Object) and entity domain data (Domain Transfer Object)

#### 5.2.5.2 How Does a Transfer Object Work?

- Encapsulates related business data
- Single method call is used to send and retrieve the Transfer Object
- Business object (EJB or factory) responsible for constructing Transfer Object
- Passed by value to client via RMI
- Must be serializable
- Immutable and mutable strategies



#### 5.2.5.3 Transfer Object Advantages

- Can simplify remote interface of EJBs
- Reduces number of get and set methods
- Improves performance
- Fewer remote calls
- Reduced network traffic
- Can access arbitrary sets of data specific to client requirements

#### 5.2.5.4 What Are the Disadvantages?

- Can introduce stale transfer objects
- Transfer objects can be cached by business objects and no longer represent current state of data
- Can increase complexity due to versioning
- Simultaneous updates must be handled

- Requires transaction isolation levels to be considered
- Inconsistent data could be read if isolation level less than TRANSACTION-SERIALIZED

#### 5.2.5.5 Why Use a Transfer Object?

- Each call to an EJB is potentially a remote call with network overhead
- Client usually requires more than one attribute of an entity
- e.g. For display purposes
- Accessing each attribute individually increases network traffic which degrades performance

#### 5.2.5.6 What Is a Session Façade?

- Defines a uniform coarse grained service access layer
- Hides the complexity of the interaction of business objects participating in a workflow
- Removes need for client to manage relationships of these business objects
- Allows consolidation of use cases



#### 5.2.5.7 How Does a Session Façade Work?

- Provides a simplified interface for business services
- Stateless or stateful strategy
- Stateless supports processes that require a single method call to complete
- Stateful supports processes that require multiple methods to complete
  e.g. The process requires conversational state

#### 5.2.5.8 Session Façade Advantages

- Improves performance by reducing network overhead
- Introduces control layer for business tier
- Centralizes security management and transaction control
- Reduces coupling of tiers

- Exposes uniform interface for interaction\?Coarse grained access to business services
- Exposes fewer remote interfaces to clients

#### 5.2.5.9 Why Use a Session Facade?

- Enforces execution of the use case in a single network call
- Provides a clear layer to encapsulate business and workflow logic used to fulfill use cases
- With EJB 2.0, session façade can use local interfaces to avoid network overhead

#### 5.2.5.10 What Is a Business Delegate?

- Object that decouples the presentation tier and business services tier
- Abstracts the underlying implementation details of the business services
- Provides domain level access point for different clients in the presentation tier to access the business services

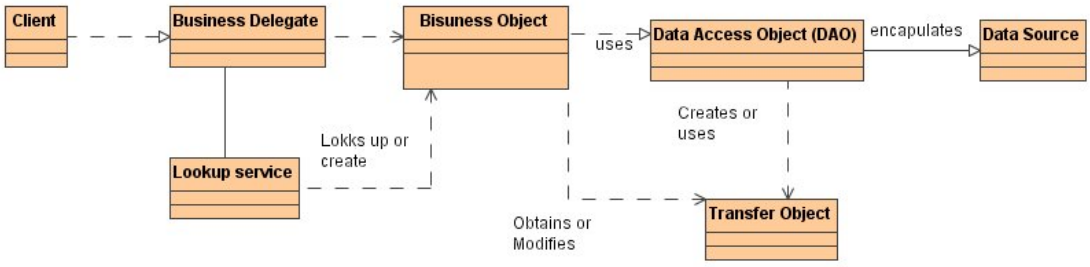


#### 5.2.5.11 How Does a Business Delegate Work?

- Encapsulates the lookup and access details of the business services e.g. EJB access details (home, remote creation and lookup)
- Handles exceptions from the services layer and translates them to business exceptions
- Transparently performs any retry or recovery operations if a service fails

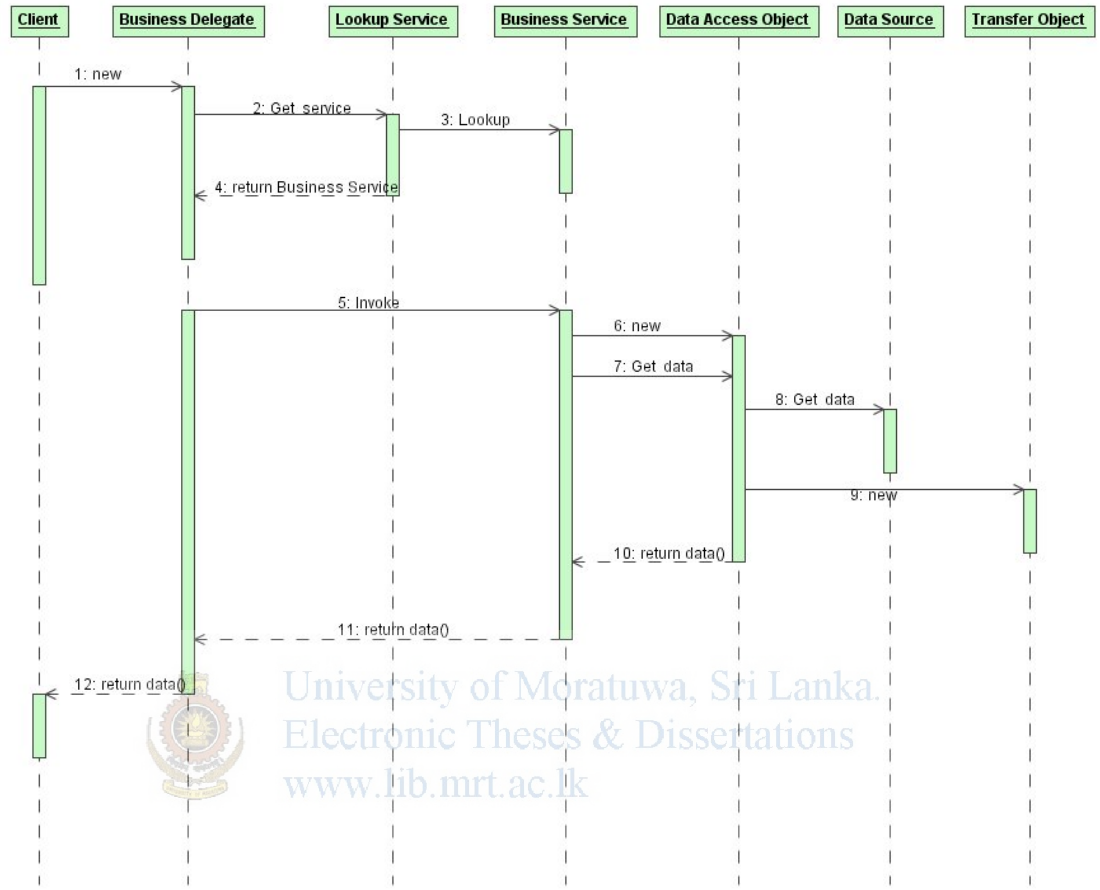
#### 5.2.5.12 Business Delegate Relationships

**Figure 5-5 Business Delegate Relationships**



### 5.2.5.13 Business Delegate Interactions

**Figure 5-6 Business Delegate Interactions**






### 5.2.5.14 Business Delegate Advantages

- Reduces coupling between tiers
- Translates business services exceptions
- E.g. Network and infrastructure exceptions translated to business exceptions
- Provides a uniform interface to the business tier
- Supports failure recovery
- Can improve performance through caching

#### 5.2.5.15 What Are the Disadvantages?

- Introduces an additional layer of abstraction
- Small amount of additional upfront work; but has considerable long term benefits
- Location transparency can hide remoteness from developers

#### 5.2.5.16 Why Use a Business Delegate?

- Allows different clients (devices, web, thick client) to uniformly access business services
- Decouples presentation tier from business tier
- Can minimize impact of business services API changes on presentation tier
- Hides underlying implementation details of services such as lookup and access
- Can perform caching of service information
- Can act as an adapter between disparate systems in a B2B system



### 5.2.6 Component Based Software Developments

Nowadays, software engineering is moving forward on an architecture-based development conception where systems are built by composing or assembling components that are often developed independently. The key to making a large variety of software products and reducing time to market is to build pieces of software where the development effort can serve in other products as well. Thus, large-grained components are becoming a practical part of an enterprise component strategy. Such generic components usually include interactions with other components, code, design models, design patterns and specifications. In addition, they must provide ways to be adaptable and customizable according with the client's needs. In this context, enterprise frameworks offer an appropriate base for waving the software architectures, components, and core requirements into one container that is adaptable, customizable, extensible, and reusable.[3]

#### 5.2.6.1 Purpose and Origin

Component-based software development (CBSD) focuses on building large software systems by integrating previously-existing software components. By enhancing the flexibility and maintainability of systems, this approach can potentially be used to reduce software development costs, assemble systems rapidly, and reduce the spiraling maintenance burden associated with the support and upgrade of large systems. At the foundation of this approach is the assumption that certain parts of large software systems reappear with sufficient regularity that common parts should be written once, rather than many times, and that common systems should be assembled through reuse rather than rewritten over and over. CBSD embodies the "buy, don't build" philosophy espoused by Fred Brooks [Brooks 87]. CBSD is also referred to as component-based software engineering (CBSE) [Brown 96a, Brown 96b].

Component-based systems encompass both commercial-off-the-shelf (COTS) products and components acquired through other means, such as no developmental items [2]



Developing component-based systems is becoming feasible due to the following:

- the increase in the quality and variety of COTS products
- economic pressures to reduce system development and maintenance costs
- the emergence of component integration technology (see Object Request Broker)
- the increasing amount of existing software in organizations that can be reused in new systems
- CBSD shifts the development emphasis from programming software to composing software systems [Clements 95].

#### 5.2.6.2 Technical Detail

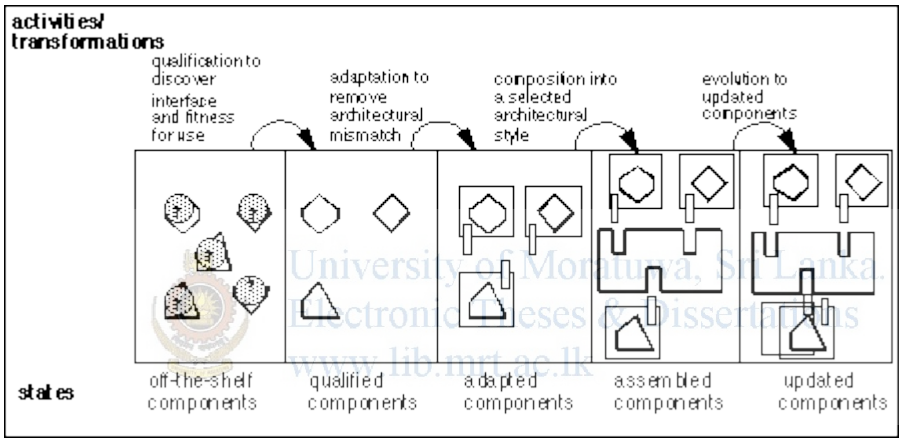
In CBSD, the notion of building a system by writing code has been replaced with building a system by assembling and integrating existing software components. In contrast to traditional development, where system integration is often the tail end of an implementation effort, component integration is the centerpiece of the approach; thus, implementation has given way to integration as the focus of system construction. Because

of this, inheritability is a key consideration in the decision whether to acquire, reuse, or build the components.

As shown in Figure 4, four major activities characterize the component-based development approach; these have been adapted from Brown [Brown 96b]:

- component qualification (sometimes referred to as suitability testing)
- component adaptation
- assembling components into systems
- system evolution

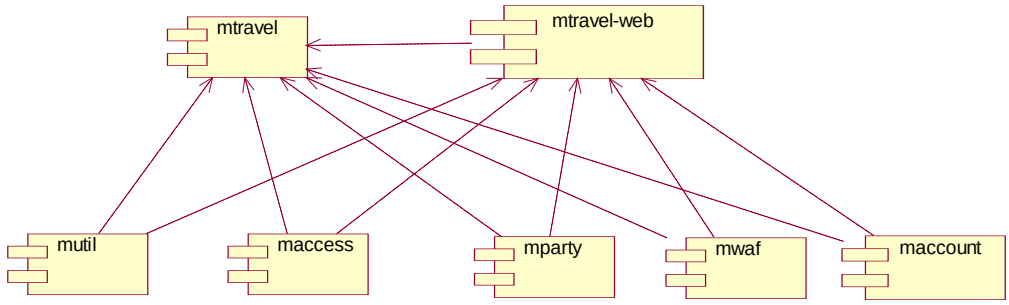
**Figure 5-7 Activities of the Component-Based Development Approach**




Each activity is discussed in more detail in the following paragraphs.

### 5.2.7 Component used in the application

**Figure 5-8 Component Used**



#### 5.2.7.1 Util component – mutil

1. This component has lot of reusable methods such as date conversion, internationalization related methods, etc. So during the development of the application I have used lot of methods available in this component, instead of rewriting those.

#### 5.2.7.2 WAF component (Web Architectural framework)– mwaf

1. This component is supported to user friendly GUIs.
2. Within the application I have extended lot of classes from base classes available in this component. It allows enabling more features in the GUI with very less work.

#### 5.2.7.3 Access component – maccess

1. This component is used to give the access to the application.
2. Access control facilitates to the following functionality of the application
    - Managing Users
    - Managing User Roles
    - Managing security access level for each role.
3. This component facilitates grant access to UI for relevant roles.



#### 5.2.7.4 Party component – mparty

1. This component is used to maintain employee details of the company.
2. Facilitates to the following functionality of the application
    - Maintain employee details
    - Maintain relationships such as Managerial, Approvers, and Reviewers.
    - Maintain company hierarchy

#### 5.2.7.5 Account component – maccount

1. This component is used to maintain company account information.

2. Since this application is related to expenses of the company it should link with the company accounting system. Therefore I have integrated the account component to the application.

### 5.2.8 Program Structure

Since the system uses MVC (Model - View - Controller) architecture, and always should starts with Controller Servlet class, which is of course the controller of the whole system.

This controller Servlet is named as '**BaseAction.class'**. This class is inherited from the '**Action.class'**, which resides in the '**org.apache.struts'** package in Struts API. Each request passing from the JSP pages, those are acting as boundary classes to the user, triggers the **process ()** method of this class.

All the JSP pages those are interacting with the user through the browser are acts as the 'View' part of the system.

The Controller part of the system is consists of Enterprise Java Bean classes, including Session Beans. This controller part is responsible for the processing of whole business logic of the system.



All together there are five main Session Beans classes are introduced to handle each major activities of the system.

- **TravelAdminBean.class -** Responsible for the business logic part in all travel admin related functionalities.
- **TravelBean.class -** Responsible for the business logic part in travel bill related activities of the system.
- **ConfigurationBean.class -** Responsible for the business logic part in all configuration activities.
- **Policy.class** – Responsible for the business logic part in agreement and policy related activities.
- **TravelReportBean.class** – Responsible for the business logic for the report generation functionalities of the system.

The **BaseAction** Servlet class does selecting the relevant session bean and execution of the business process connected with the user request. After executing such business logic, it is the responsibility of the mentioned session bean to pass the response to the user.

Apart from the above-mentioned Session Bean classes, the system is consisting of DAO layer, is responsible for accessing all the data base activity. Some of those DAO classes are

- **OracleTravelDAO.class**
- **OracleTravelAdminDAO.class**
- **OraclePolicyDAO.class**
- **OracleReportDAO.class**

## 5.3 Summery



System implementation is discussed covering architecture of the system. Prior to implementation evaluation of the common architecture suitable for such a system also taken into discussion. Suitable tools used for implementing are listed. Major code fragments and important techniques are also discussed in detail.

Following chapter will discuss about the testing.