Chapter 5 Framework Implementation

The library systems domain application framework is implemented according to the system analysis and design models in relation to the layered system architecture of (Jacobson, 1997).

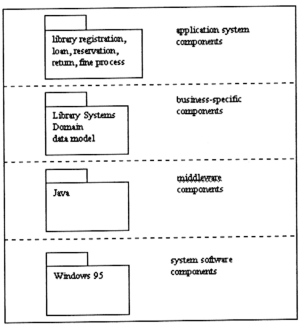

Figure 5.1: Layered System Architecture

Based on Figure 5.1 which displays the reusable library software system in a layered architecture, the Window 95 operating system is the choice of implementation platform at the system software layer. The reason for choosing Windows operating system is not a major factor but simply of widespread availability. The immediate upper layer is the middleware layer which is omnipresent in local and

distributed systems. The programming language, Java, is the preferred choice for this project's middleware layer implementation for its object-oriented features, platform-independent features and useful utility packages and classes such as the Java Collection classes and Enterprise Edition System. Although an independent application is developed and implemented in this project to test the viability of the library systems domain application framework, Java has the necessary Application Programming Interfaces (API) such as Remote Method Invocation (RMI) and networking packages for distributed system development. This is left out for future improvements.

On top of the middleware layer is the business-specific layer which consists of the library system domain model and data management packages. These packages consist of software components that make up a significant part of the framework and development effort put into this research project for the library systems domain. The typical scenarios of library entities' interactions and states as modelled in the dynamic interaction diagrams are implemented into the business-specific layer. The library packages implementation is based on the "Model-Controller-View" (MVC) design pattern that identifies three communicating packages which are the domain model, data management and user interface packages. The reusable common library components, such as the library system domain model and subsystems of these packages are placed here. Basically, these library systems domain packages are deployed to application developers for framework utilisation, customisation and application development.

Lastly, the implementation for the topmost layer, which is the application systems layer, consists of the library software components with specific functions in the

application such as registration of borrowers and inventory of titles, loan, reservation, return of loaned item and fine on late return. The implementation coordination which comprised of activities' logic control flow between classes in these packages is important to fulfil the library systems domain requirements of an application system. However, a library application system offers little reuse and extensibility in framework development because of the specific application system functions. The purpose of this application systems layer is to develop specific application systems from components customised by application developers that fit the needs of an end-user library system. As part of the implementation, a default application system is developed for framework testing purpose and immediate utilisation of library features by application developers.

The variabilities of library requirements and implementations were identified as variation points left to the application developers to customise and are based on some abstract factory methods or common patterns identified in the initial framework analysis and design. This project's implementation of an initialisation file for configuration of library entity status and business rules will fulfil these variation points.

5.1 Implementation Classes

The use of a framework necessitates intimate knowledge of the framework implementation and its classes. This type of framework is better known as a white-box framework. Based on this consideration, this research project on library systems domain application framework was set out to be implemented in such a fashion.

The library systems domain application framework has a programming model that is made up of a common generic base classes in the structure of reusable software

component like JavaBeans. By modelling these classes as base factory classes and let application developers extend derived classes with additional methods and attributes to create customised objects, we can ensure a high degree of structural consistency across applications and maintain a common set of indirect object instantiations to avoid direct manipulation and mistaken changes to the library domain classes.

The library systems domain application framework has implementation classes developed using Java's JDK 2 language paradigm with powerful object-oriented features. The UML high-level system design models comprise of several analysis types and subsystems with relationships to define the library systems domain architectural structure. Therefore, the models have significantly assisted implementation effort of the library domain requirements and action rules. The necessary functions of a library application are provided at the source code level in classes that exhibit high cohesion. Library domain objects will call on the functions at the implementation level, such as, functions with searching or sorting algorithm, database access and graphical functions. Most of the implementation classes which exist in a component subsystem are organised in a package as part of the application framework. These classes are placed within the same level with a primary aim of software reuse but they are distinguished on the basis of a domain model. Classes within a package component have relationships to function as a subsystem.

5.2 Configuration and Initialisation of Business Rules

An utility class which accesses business rules in a configuration file as shown in Figure 5.2 is set to pre-defined properties in a hash table data structure of String

values called HashMap in Java language. It is implemented to synchronise with information such as maximum period of a loan, the number of loans for each member and maximum days of reservation period that applies towards the library system. For example, this utility class named "Status" with static methods is used by the library's Borrower class to look up the persistent status of a borrower to ensure he or she does not exceeds the maximum number of loans. The Copy class also calls on the methods in the utility class to confirm on the availability of a copy item. A single configuration file can quickly update the pre-conditions, initialisation and validation of a business rule. A set of pre-conditions is usually imposed to different groups of library members, for example, a registered member has a different set of rules or privileges from a non-registered member, or in a university library, a student has less privileges than a working staff.

```
available=available
borrowed=borrowed
onhold=onhold
allout=all borrowed
copyReturned=copy returned
reserved=reserved
MaxReserve=5
MaxLoan=5
dbName=poet://LOCAL/my_base
fineDayWeek=25
fineDayMonth=100
DueCondition1=7
DueCondition2=14
loanDays=14
holdDays=7
membership=staff,member,public
stafffee=1
memberfee=10
publicfee=20
```

Figure 5.2: Initialisation File for Configuration of Library Entity Status and Business rules

As shown in Figure 5.2, an editable text-based file is used for default initialisation or updates to the parameter variables representing the library systems domain requirements. During the application system run-time and using the hashmap algorithm, a referenced parameter is read as a key and the assigned value is returned. Hence, this value is passed as a message to domain class constructors or abstract factory methods in the control logic classes. Dynamic object instantiations such as Loan, Reservation and Borrower are then possible. The dynamic state of library entities can be identified as an internationalisation variable in the initialisation file for database updates and persistency. The state of a library entity is important to be tracked down for search and retrieval of loan or reservation, such as in a status of available, borrowed or onhold.

The initialisation file can be overwritten with different business rules simply by referring to the parameter key and reassigning a new value. For example, the Loan period allowed to a member is referred with a parameter key named "loanDays" is assigned 14 days and this can be changed to suit a new rule.

5.3 Storage Management

The storage management factor is considered to be one of the important implementation issues. Absolute transient data memory is impractical because data entered cannot be saved permanently for future references. Data storage management is a major factor for library system requirements especially for research projects. The data can be complex as it involves composition, association, aggregation and other relationships, especially with an object-oriented database.

Our decision to opt for a Java-based object-oriented database called POET (POET, 1998) is an important implementation issue for proving the benefits it offers. The full potential of an object-oriented database is yet to be discovered but the significant advantages of object-oriented database technology in general promotes its use. The benefits of POET implementation are its Object Query Language (OQL), persistent-aware classes, utility classes, collection-persistent classes with typical data structures as well as its algorithms and synchronisation methods to avoid transaction collisions.

Relational Database Management System (RDBMS) are very popular among business domains due to the fact that the relational database model is useful for heavy data, light on relationships (Morgan, 1998) and has existed for a long time. The adoption of an object-oriented framework for implementation and integration into an existing library system using a legacy or relational database is a major consideration. An application framework based on object-oriented technology is still considered quite risky to business although beneficial and expects a slow change-over and transition from an old system to a new system. However, an object-oriented database is useful for persisting data and relationships among objects and outweighs a relational database in a seamless integration with an object-oriented implementation language.

Iteration of data objects in the storage is a pre-requisite design pattern for the collection class. Each base class of the framework will find iteration useful to retrieve the searchable object in the underlying data model and structure. According to (Booch, 1994), by isolating the patterns of storage management, we can produce a robust yet adaptable library. The design pattern should decouple storage

management policy from its implementation. A framework design should follow such a pattern. The choice of database implementation and its operations should not be restricted by the framework design.

5.4 Application Framework Customisations

By creating a wizard tool and adopting it as part of the framework to help application developers use the framework quickly and realise the many benefits the framework offers to efficient software development. A wizard tool can lead the framework user through steps of package selections and feature customisations, which at the end of it, can produce a working application without a single inclusion of coding by the application developers. An application framework can be difficult to adopt quickly for application development due to its complexity and size. The idea of the wizard tool is borrowed from the program installation setup package which now becomes the favourite way of software deployment. The windows interface of a wizard tool will offer a friendly interface for selection options and customisation of the framework components.

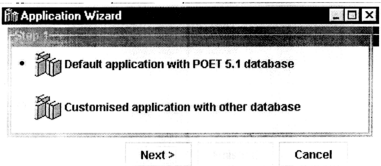

Figure 5.3: Step 1 of Wizard tool –
Choosing the Framework's Application Database.

Figure 5.3 shows a window's interface of the wizard tool as the first step to assist framework users in a selection on the use of database with the framework components. The first option is the default library system domain application developed with the use of POET 5.1 object-oriented database. Application developers can immediately use the application without further development if they choose this option. The second option on the use of framework is to choose a preferred database to go with it. Application developers can choose to develop and customise their application, and yet, still use the framework components. As appeared below the figure, the "next" button is enabled to lead the framework user to the next step on the use of a wizard tool.

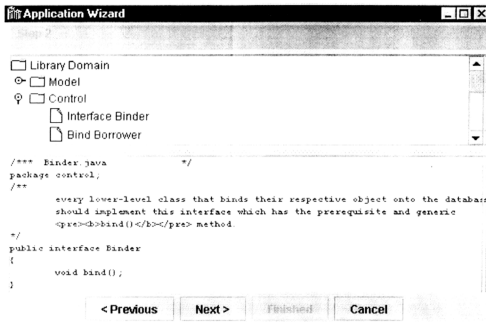

Figure 5.4: Step 2 of Default Option in Step 1 Wizard tool

In Figure 5.4, the upper portion of the window shows a tree hierarchy of framework packages and contents. As shown in the figure, by choosing a particular package component such as "Control" will reveal all its interface and classes. By further selecting the interface, the implementation details at the source code level are revealed. Application developers can refer to this code for understanding and development. The proper ordering of components into packages will quickly assist application developers in the reuse of the framework.

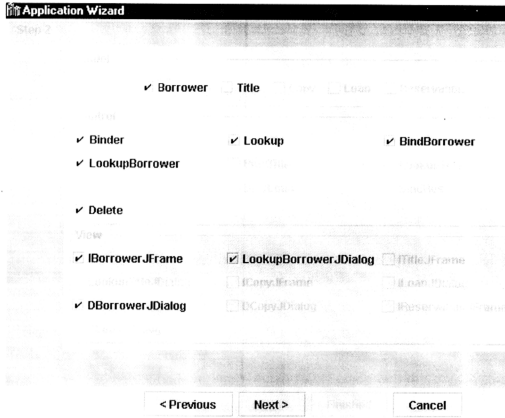

Figure 5.5: Step 2 of Customise Option in Step 1 Wizard Tool

Based on Figure 5.5, when application developers opt to customise his or her own application by using the framework components, a screen is provided for the selection of classes from each packaged component. Consequently, the wizard will act as an automated code generation tool and generate the relevant component and its selected classes. Only related classes and components can be enabled for selection based on association and message dependency.

5.5 Application Framework Evaluation

To ensure the success of a framework, we must test it by building different applications from it in the related problem domain. By starting off with a small framework, we can manage it easier and eventually expand it by hooking it up with other small frameworks. A framework should concentrate on a specific problem domain because we do not want to build a class library. Lastly, we encourage other developers to use our framework and develop extension frameworks.

For components and subsystems to interact, we have to test each unit first before testing the interactions between the different units. We can test each subsystem as a packaged component unit, for example, a Loan subsystem or a Reservation subsystem. Each component has to be tested to ensure a framework is reliable and reusable. The interface facade between subsystems, for example, between the Loan, the Borrower and the Reservation, would have to be compliant and efficient for high degree of reusability. A component testing can determine the stability of the framework in order to have a reliable way of reusing one or more components.

Continuous iterations of improving the framework design over time can help increase the degree of components reusability. Unforeseen demands from application developers can be included if the framework is put to use and tested.

As mentioned earlier during system design, the library system domain components are designed with interfaces or facades to facilitate reuse and avoid complex framework implementation details. As such, the implementation of each generic domain class with its respective interface and default implementation is subclassed with preferred implementation for application development. For example,

class Loan is derived from a generic and default class called SuperLoan that does not include practical methods implemented with specific library rules.

The default library application system developed with the object-oriented application framework for library systems domain can be installed as described in Appendix I: Installation Guide, and used as described in Appendix II: User Guide.

The following screen captures are from the default library application system. The tested sequence of activities are closely matching the designed interaction diagram, namely the sequence diagram.

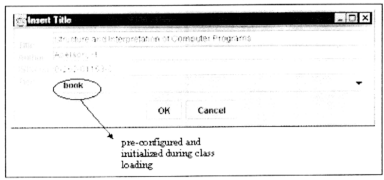

Figure 5.6: Insert a Title Dialog

According to Figure 5.6, insertion of a Title object into the library system requires attributes of Title's name, author's name, ISBN no. and type of item. The item may be a book, magazine, CD, video or others. The items can be manually set in a configuration file based on a hash table of strings key and value pair mapping. With the existence of different library systems, application developers using the library systems domain application framework can customise the variable features because the framework was designed after careful domain analysis, with such variabilities in mind. This will only promote the value of framework reusability.

If additional attributes of a Title object is needed besides those mentioned in the Title dialog, the application developer can overload the class constructor or extend the base class and derive a specific constructor. The additional features of the front-end dialog is added to the default base class's generated code from the application framework.

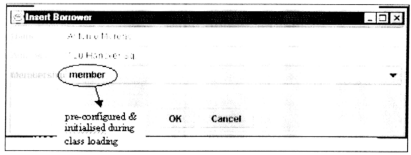

Figure 5.7: Insert a Borrower Dialog

A dialog box for inserting a Borrower into the library system is shown in Figure 5.7. The attributes expected of a Borrower object are name, address and type of membership. The membership status, such as member, staff or public, is set as a key and value hashmap in a configuration file.

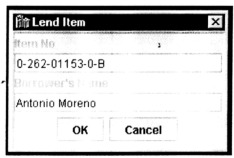

Figure 5.8: Insert a Loan Dialog

reservation. The included details of a Reservation are title, date and time of reservation which are circled in the figure.

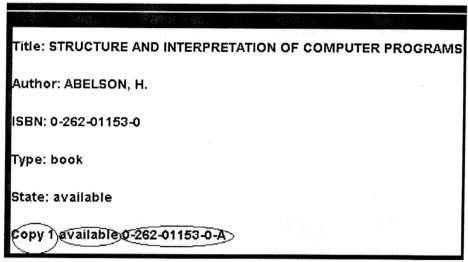

Figure 5.10: Information Retrieval of a Title.

In Figure 5.10, the window reflects an information search on class Title. Based on relative search, we can retrieve one or more records that match. The Title details included are title, author, ISBN number, type of item, availability state, and lastly, the last line shows the details of an aggregate copy, such as number copy, status and item identification number, which are circled in the figure.

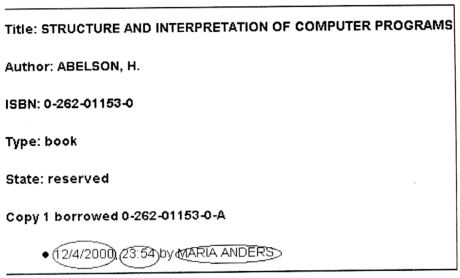

Figure 5.11: Information on a Title object with Reservation Status

As shown in Figure 5.11, the search on a Title can produce the current availability status as reserved if all copies are on loaned and a Reservation is placed on it. The above window displays the information of a single copy which is borrowed on the particular date, time and by the borrower which are circled in the figure.

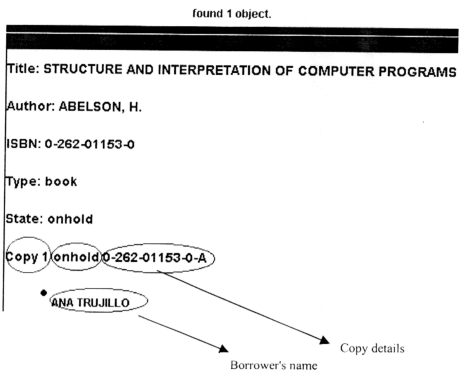

Figure 5.12: Onhold Status for Reservation.

When we look up a Title and if a Reservation is placed on the Title, the immediate return of the Title's copy will be rendered with a pending (onhold) status as shown in Figure 5.12. The Reservation details in the figure include title, author, ISBN number, type of item, availability state, the details of the copy reserved, such as, copy number, status, item identification number which are circled in the figure, and the Borrower's name, which is circled in the figure, who is the first to get the book.

We can conclude from a library scenario of searching for an item such as a book, magazine, video, or CD, the tasks of looking for a unique attribute, such as an item catalogue identification number, will provide us with an access and indexed key

which maps to other relevant attributes of an object such as type of item, the title, the availability state and the author. The access key is an argument in the object query statement of OQL. If there are multiple copies of a Title, all the copies details will also be displayed as a generated list from a query traversal of an aggregate entity type. The list's instances display unique identifiers and typically hold references to other entities of domain classes with respective *get* and *set* methods.

The library rules have a chain of responsibility events in action, as depicted in a state diagram or a sequence diagram, to identify the state of an object before an operation is carried out on that object. For example, the Loan policy on a Copy is based on the availability, reservation list, and eligibility of loans allowed for each Borrower. The relevant information affecting an object is retrieved and displayed upon query. This chain of responsibility driven policy is a design pattern adopted for implementation (Gamma, 1994). In this library application system, a library rule is placed on the number of loans allowed to a Borrower, which is a dynamic value in the initialisation file easily configurable for run-time.

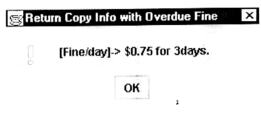

Figure 5.13: Fine on Borrower for overdue Loan.

As shown in Figure 5.13, upon the return of a Copy on Loan, the system will determine the last returning date. If it is overdue, a fine will be imposed on each day

based on pre-configured key/value pair. For example, the system is set to a $0.25 fine for a day and the system will return $0.75 fine for 3 days.

The time period of late return is defined into two stages in the library system application prototype, DueCondition1 and DueCondition2, with fines of 25 and 100 monetary units respectively, i.e. the imposition of 25 unit for the overdue of first 7 days and 100 unit for the following 7 days. Such a library system policy is specific to a library application and is dynamically associated with an entity through the initialisation file at run-time. This concept is based on the Strategy Pattern defined in (Gamma, 1994).

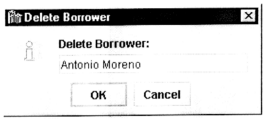

Figure 5.14(a): Delete a Borrower Dialog

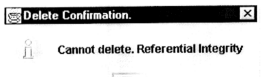

Figure 5.14(b): Information Feedback of Deletion on Borrower

According to Figure 5.14(a), a window dialog is provided to delete the Borrower's record from the database. After entered the Borrower's name into the dialog and pressed the "OK" button, the Figure 5.14(b) window dialog appeared as

an information feedback if the Borrower has an outstanding Loan or Reservation, deletion is not permitted due to data relationship and referential integrity between the entity classes of Borrower and Loan or Reservation. The referred borrower still has outstanding loans or reservations. This scenario illustrates the concept of "Ownership". The implementation of object ownership as an aggregate attribute of an entity type persists the objects and its relationships. The deletion of the container object does not trigger an immediate deletion of its contained reference object which may be a collection of object references. This application design reflects an important business rule on entity relationships and data persistency. Additionally, a feature of the POET Object-Oriented Database Management System (OODBMS) reflects such an underlying implementation in regards to object deletion and ownership.

5.6 Summary

The library systems application framework is being tested from its implementation as a prototype application. After implementing the framework architecture as a layered system, we tested it by interfacing the library application system components to the underlying library-specific systems domain components, middleware components and system software components. We tested each unit application and component system by itself and also as part of the layered system. The interoperability of the components acting as a packaged subsystem is important to ensure the high degree of reusability of these components.

The interaction diagrams in the system design phase has given us the necessary guidelines in implementing the library rules and activities to fulfil the library

application system development. The state and attributes of an object type are parameters affecting an operation.

The reusability features of a library systems domain application framework were tested by incorporating configurable items parameters in an editable template file. Variations points can then be adopted with ease in different library application systems while still retaining the reusable library systems domain components. The biggest factor affecting the usability of the library systems application framework is the option to choose the type of database to work with, which could be a legacy database or a new type of database, such as an OODBMS. Lastly, the wizard tool is a convenient way to quickly assist application developers in producing a library system application with little or no coding.